TCP/IP - WORLDFIP SUITE
Software User Manual

ALS 53425 a-en

First issue: 09-2000

Meaning of terms that may be used in this document / Notice to readers

WARNING

Warning notices are used to emphasize that hazardous voltages, currents, temperatures, or other conditions that could cause personal injury exist or may be associated with use of a particular equipment.

In situations where inattention could cause either personal injury or damage to equipment, a Warning notice is used.

Caution

Caution notices are used where there is a risk of damage to equipment for example.

Note

Notes merely call attention to information that is especially significant to understanding and operating the equipment.

This document is based on information available at the time of its publication. While efforts have been made to be accurate, the information contained herein does not purport to cover all details or variations in hardware or software, nor to provide for every possible contingency in connection with installation, operation, or maintenance. Features may be described herein which are not present in all systems. ALSTOM assumes no obligation of notice to holders of this document with respect to changes subsequently made.

ALSTOM makes no representation or warranty, expressed, implied, or statutory with respect to, and assumes no responsibility for the accuracy, completeness, sufficiency, or usefulness of the information contained herein. ALSTOM gives no warranties of merchantability or fitness for purpose shall apply.

In this publication, no mention is made of rights with respect to trademarks or tradenames that may attach to certain words or signs. The absence of such mention, however, in no way implies there is no protection.

Partial reproduction of this document is authorized, but limited to internal use, for information only and for no commercial purpose.

However, such authorization is granted only on the express condition that any partial copy of the document bears a mention of its property, including the copyright statement.

All rights reserved.
© Copyright 2000. ALSTOM (Paris, France)

Index letter	Date	Nature of revision

Revisions

1. CONTENT OF THIS MANUAL

Chapter 1: **Introduction:** Short description of the composition of the TCP/IP –WorldFIP Suite.

Chapter 2: **General overview of the software:** Describes the source code directories of the software package and the user interface (socket functions, TCP/IP Stack functions) and the creation of an application.

Chapter 3: **Software configuration**: Describes the configuration parameters (selection of the target machine operating system, definition of the stack services, setting of the operation parameters of the active services) and the RAM allocated by the TCP/IP Stack and the WorldFIP pseudo-driver.

Chapter 4: **Operating system adaptation:** Describes the adaptation of the TCP/IP Stack and the Tool Box module to the operating system.

Chapter 5: **TCP/IP Sequencer:** Describes the TCP/IP Sequencer module and gives examples for use in a multi-task context and a mono-task context.

Chapter 6: **Device driver adaptation:** Describes the functions for the adaptation of the device driver and gives some examples for these functions.

Chapter 7: **Software startup and stoppage:** Describes the startup and stoppage procedures of the software. For each procedure you will find the inputs/outputs and an example.

Chapter 8: **Socket functions:** Describes the socket functions for clients and servers that use stream sockets and for clients and servers that use datagram sockets.

Chapter 9: **TCP/IP Stack module functions:** Describes the functions of the TCP/IP Stack module with their input and output parameters and gives examples for these functions.

Chapter 10: **Processing the real time clock interrupt**

Appendix A: **Example of a stream socket client**

Appendix B: **Example of a stream socket server**

Appendix C: **Example of a datagram socket client**

Appendix D: **Example of a datagram socket server**

2. RELATED PUBLICATIONS

ALS 50278 FIP DEVICE MANAGER Software Version 4 User Reference Manual

ALS 50249 FIP Network general introduction

3. WE WELCOME YOUR COMMENTS AND SUGGESTIONS

ALSTOM strives to produce quality technical documentation. Please take the time to fill in and return the "Reader's Comments" page if you have any remarks or suggestions.

Preface

Reader's comments

ALS 53425 a-en	***TCP/IP - WORLDFIP SUITE Software User Manual***

Your main job is:

- ❑ System designer
- ❑ Distributor
- ❑ System integrator
- ❑ Installer
- ❑ Programmer
- ❑ Maintenance
- ❑ Operator
- ❑ Other (specify below)

If you would like a personal reply, please fill in your name and address below:

COMPANY:.....NAME:.....

ADDRESS:.....

.....COUNTRY: .....

Send this form directly to your ALSTOM sales representative or to this address:

ALSTOM Technology
Technical Documentation Department (TDD)
23 - 25 Avenue Morane Saulnier
92360 – Meudon la Forêt
France
Fax: +33 (0)1 46 29 12 01

All comments will be considered by qualified personnel.

REMARKS

Reader's comments

Contents

CHAPTER 1 - INTRODUCTION

CHAPTER 2 - GENERAL OVERVIEW OF THE SOFTWARE

1. SOFTWARE PACKAGE 2-1
   - 1.1 SockAPI module 2-2
   - 1.2 TCP/IP Stack module 2-3
   - 1.3 WorldFIP pseudo-device driver module 2-7
   - 1.4 Tools Box module 2-8
   - 1.5 Example 2-10
2. SOFTWARE CONFIGURATION 2-11
3. OPERATING SYSTEM ADAPTATION 2-11
4. TCP/IP SEQUENCER 2-11
5. DEVICE DRIVER ADAPTATION 2-12
6. USER INTERFACE 2-12
   - 6.1 Socket functions 2-12
     - 6.1.1 Standard services: 2-13
     - 6.1.2 Non standard services: 2-13
   - 6.2 TCP/IP Stack specific functions 2-14
7. APPLICATION CREATION 2-15
   - 7.1 Libraries 2-15
   - 7.2 How to create an application that uses the TCP/IP - WorldFIP Suite package: 2-16

CHAPTER 3 - SOFTWARE CONFIGURATION

1. SELECTION OF THE TARGET MACHINE'S OPERATING SYSTEM 3-2
2. ACTIVATION OF THE TCP/IP STACK'S OPTIONAL SERVICES 3-2
3. SET THE VALUE OF THE TIMER TICK 3-3
4. SELECTION OF THE WORLDFIP PSEUDO DRIVER 3-3
5. GENERIC NETWORK INTERFACE CONFIGURATION PARAMETERS 3-4
6. IP PROTOCOL CONFIGURATION PARAMETERS 3-4
7. ARP PROTOCOL CONFIGURATION PARAMETERS 3-5
8. ICMP PROTOCOL CONFIGURATION PARAMETERS 3-5
9. UDP PROTOCOL CONFIGURATION PARAMETERS 3-5
10. TCP PROTOCOL CONFIGURATION PARAMETERS 3-6
11. ALLOCATED RAM MEMORY 3-7
    - 11.1 RAM memory allocated by the TCP/IP Stack module 3-7
    - 11.2 RAM memory allocated by the WorldFIP pseudo driver 3-8

Contents

CHAPTER 4 – OPERATING SYSTEM ADAPTATION

1. ADAPTATION OF THE TCPIP_STACK MODULE 4-1
2. ADAPTATION OF THE TOOLS BOX MODULE 4-2

CHAPTER 5 - TCP/IP SEQUENCER

1. MODULE DESCRIPTION 5-1
2. EXAMPLES 5-3
   - 2.1 Example of a sequencer in a multi-task context 5-3
   - 2.2 Example of a sequencer in a mono-task context 5-5

CHAPTER 6 - DEVICE DRIVER ADAPTATION

1. OVERVIEW 6-1
2. FUNCTIONS 6-3
   - 2.1 DEV_CONTROL(): control operations 6-3
     - 2.1.1 Description 6-3
     - 2.1.2 Inputs 6-3
       - 2.1.2.1 Description of the FIP_MSG_INIT structure 6-4
       - 2.1.2.2 Description of the LSAP_DESC structure 6-6
       - 2.1.2.3 Description of the ETH_MSG_INIT structure 6-6
     - 2.1.3 Outputs 6-6
   - 2.2 DEV_WRITE(): sending packets 6-7
     - 2.2.1 Description 6-7
     - 2.2.2 Inputs 6-7
     - 2.2.3 Outputs 6-9
   - 2.3 TCPIP_RCVPACK(): receiving packets 6-10
     - 2.3.1 Description 6-10
     - 2.3.2 Inputs 6-11
     - 2.3.3 Outputs 6-11
   - 2.4 Examples 6-12
     - 2.4.1 Interfacing an Ethernet driver 6-12
     - 2.4.2 Interfacing an Ethernet and two ALSTOM WorldFIP pseudo drivers 6-16

CHAPTER 7 – SOFTWARE STARTUP AND STOPPAGE

1. TCIP_START(): STARTUP PROCEDURE OF THE TCP/IP STACK MODULE 7-1
   - 1.1 Description 7-1
   - 1.2 Inputs 7-1
   - 1.3 Outputs 7-7
   - 1.4 Example 7-8
2. SOCKAPI_START(): STARTUP PROCEDURE OF THE SOCKAPI MODULE 7-14
   - 2.1 Description 7-14
   - 2.2 Inputs 7-14
   - 2.3 Outputs 7-14
   - 2.4 Fault messages 7-14
   - 2.5 Performance 7-14
3. TCPIP_STOP(): STOPPAGE PROCEDURE OF THE TCP/IP STACK MODULE 7-15

4. SOCKAPI_STOP(): STOPPAGE PROCEDURE OF THE SOCKAPI MODULE....7-16
4.1 Description....7-16
4.2 Outputs....7-16
4.3 Fault messages....7-16
4.4 Performance....7-16

CHAPTER 8 - SOCKET FUNCTIONS

1. ANOMALY REPORTS....8-4
2. BASIC STRUCTURES....8-5
3. SOCKET()....8-6
3.1 Description....8-6
3.2 Inputs....8-6
3.3 Outputs....8-6
3.4 Fault messages....8-7
3.5 Performance....8-7
4. BIND()....8-8
4.1 Description....8-8
4.2 Inputs....8-8
4.3 Outputs....8-8
4.4 Fault messages....8-8
4.5 Performance....8-8
5. SOCKAPI_IOCTL()....8-9
5.1 Description....8-9
5.2 Inputs....8-10
5.3 Outputs....8-10
5.4 Fault messages....8-11
5.5 Performance....8-11
6. CONNECT()....8-12
6.1 Description....8-12
6.2 Inputs....8-12
6.3 Outputs....8-12
6.4 Fault messages....8-13
6.5 Performance....8-13
7. LISTEN()....8-14
7.1 Description....8-14
7.2 Inputs....8-14
7.3 Outputs....8-14
7.4 Fault messages....8-14
7.5 Performance....8-15
8. ACCEPT()....8-16
8.1 Description....8-16
8.2 Inputs....8-16
8.3 Outputs....8-16
8.4 Fault messages....8-17
8.5 Performance....8-17

Contents

9. SEND() 8-18
9.1 Description 8-18
9.2 Inputs 8-18
9.3 Outputs 8-19
9.4 Fault messages 8-19
9.5 Performance 8-19
10. SENDTO() 8-20
10.1 Description 8-20
10.2 Inputs 8-21
10.3 Outputs 8-21
10.4 Fault messages 8-21
10.5 Performance 8-22
11. RECV() 8-23
11.1 Description 8-23
11.2 Inputs 8-24
11.3 Outputs 8-24
11.4 Fault messages 8-25
11.5 Performance 8-25
12. RECVFROM() 8-26
12.1 Description 8-26
12.2 Inputs 8-26
12.3 Outputs 8-27
12.4 Fault messages 8-27
12.5 Performance 8-27
13. CLOSESOCKET() 8-28
13.1 Description 8-28
13.2 Inputs 8-28
13.3 Outputs 8-28
13.4 Fault messages 8-28
13.5 Performance 8-28
14. SOCKAPI_GET_LAST_ERROR() 8-29
14.1 Description 8-29
14.2 Outputs 8-29
14.3 Performance 8-29
15. SOCKAPI_IP2DOT() 8-30
15.1 Description 8-30
15.2 Inputs 8-30
15.3 Performance 8-30
16. SOCKAPI_DOT2IP() 8-31
16.1 Description 8-31
16.2 Inputs 8-31
16.3 Performance 8-31

CHAPTER 9 - TCP/IP STACK MODULE FUNCTIONS

1. TCPIP_GETADDR(): REQUESTING THE IP ADDRESS OF A LOCAL INTERFACE 9-1
1.1 Description 9-1
1.2 Input parameters 9-1
1.3 Output parameters 9-1
1.4 Returned values 9-2
2. TCPIP_PUTADDR(): SETTING THE IP ADDRESS OF A LOCAL INTERFACE 9-3
2.1 Description 9-3
2.2 Input parameters 9-3
2.3 Output parameters 9-3
3. TCPIP_CONF_TARP(): STATIC CONFIGURATION OF THE IP ADDRESS AND HARDWARE ADDRESS CORRESPONDENCE TABLE 9-4
3.1 Description 9-4
3.2 Input parameters 9-4
3.3 Output parameters 9-4
3.4 Returned values 9-5
3.5 Example 9-5
4. TCPIP_RTADD(): ADDING AN ELEMENT TO THE ROUTING TABLE 9-8
4.1 Description 9-8
4.2 Input parameters 9-8
4.3 Output parameters 9-8
4.4 Returned values 9-9
5. TCPIP_RTDEL(): DELETING AN ELEMENT FROM THE ROUTING TABLE 9-10
5.1 Description 9-10
5.2 Input parameters 9-10
5.3 Output parameters 9-10
5.4 Returned values 9-10
6. TCPIP_ECHO_RQ(): SENDING AN ECHO REQUEST 9-11
6.1 Description 9-11
6.2 Input parameters 9-11
6.3 Output parameters 9-12
6.4 Returned values 9-12
6.5 Example 9-13
7. TCPIP_GET_SUBNETMASK(): GETTING THE SUBNET MASK 9-14
7.1 Description 9-14
7.2 Input parameters 9-14
7.3 Output parameters 9-14
7.4 Returned Values 9-14
7.5 Example 9-15

CHAPTER 10 – PROCESSING THE REAL-TIME CLOCK INTERRUPT

1.1 Introduction 10-1
1.2 Example 10-1

Contents

APPENDIX A - EXAMPLE OF A STREAM SOCKET CLIENT

APPENDIX B - EXAMPLE OF A STREAM SOCKET SERVER

APPENDIX C - EXAMPLE OF A DATAGRAM SOCKET CLIENT

APPENDIX D - EXAMPLE OF A DATAGRAM SOCKET SERVER

Figures

Figure 1.1 - TCP/IP-WorldFIP Suite composition 1-1

Figure 3.1 - Selection of the operating system 3-2
Figure 3.2 - Service Selection 3-2
Figure 3.3 - Timer Tick Value 3-3
Figure 3.4 - Selection of the WorldFIP pseudo driver 3-3
Figure 3.5 - Generic network interface configuration parameters 3-4
Figure 3.6 - IP Protocol Configuration Parameters 3-4
Figure 3.7 - ARP Protocol Configuration Parameters 3-5
Figure 3.8 - ICMP Protocol Configuration Parameters 3-5
Figure 3.9 - UDP Protocol Configuration Parameters 3-5
Figure 3.10 - TCP Protocol Configuration Parameters 3-6

Figure 4.1 - Tools Box module: Configuration of the tbx_opt.h file 4-2

Figure 5.1 - TCP/IP Stack module Sequencer 5-1
Figure 5.2 - Implementation of a sequencer in a multi-task context 5-4
Figure 5.3 - Implementation of a sequencer for a mono-task context 5-6

Figure 6.1 - Device Driver Adaptation 6-1
Figure 6.2 - FIP_MSG_INIT structure 6-4

Figure 7.1 - Definition of the Ethernet interface 7-8
Figure 7.2 - Definition of the first WorldFIP interface 7-9
Figure 7.3 - Definition of the second WorldFIP interface 7-10

Figure 8.1 - State transactions for setting up clients and servers that use stream sockets 8-1
Figure 8.2 - Transactions for setting up clients and servers that use datagram sockets 8-3
Figure 8.3 - Basic structures of the socket functions 8-5

Tables

Table 4.1 – Tools Box module: Configuration of the tbx_os.c file 4-2

Table 5.1 – Sequencer in a mono-task context (functions) 5-5

Table 6.1 – Elements of the FIP_MSG_INIT structure 6-5

Chapter 1 Introduction

The ALSTOM TCP/IP - WorldFIP Suite offers users an Internet connection over a WorldFIP network.

All standard protocols (TCP, UDP, IP, ICMP, ARP and RARP) have been developed with portability in mind, so as to be independent of the installation machine's operating system.

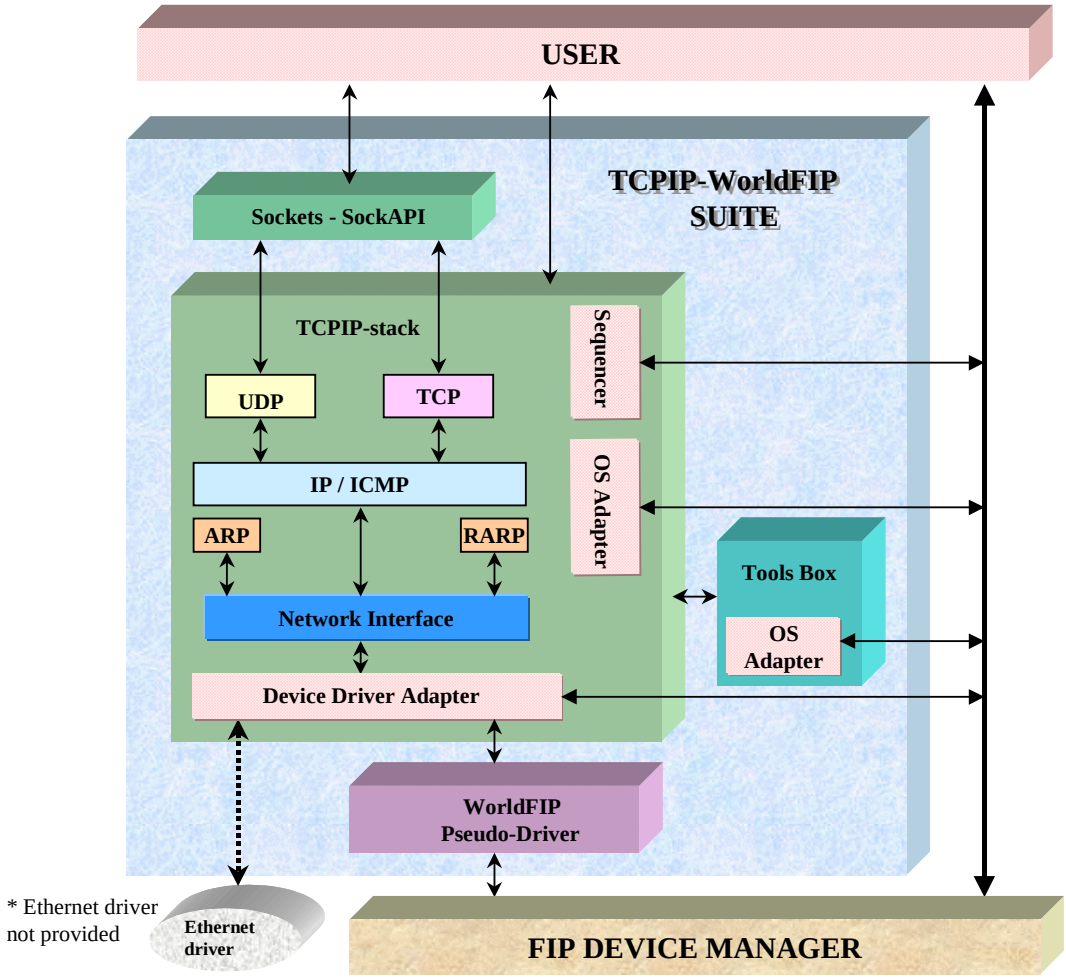


Figure 1.1 – TCP/IP-WorldFIP Suite composition

The software is delivered as a source code portable to any operating system. It has been designed for use in all contexts, from real time operating systems on embedded targets to the most common operating system (Windows NT, UNIX, etc.).

The hardware environment should have a programmable timer that produces periodic interrupts. The user has to connect the interrupt vector to the interrupt handler. This operation depends of the user hardware environment and operating system.

At the application level the software provides a network programming interface which is based on the *'socket'* paradigm popularised in Berkeley Software Distribution (BSD) from the University of California.

The access to a WorldFIP network is provided via a pseudo-driver mapped on the ALSTOM FIP DEVICE MANAGER library.

If the user needs to access an Ethernet network then the connection to an Ethernet driver (not provided by the present package) can be easily created using the Device Driver Adapter functions.

Note

The TCP/IP stack can act as a multi-host or as a router on a WorldFIP and/or Ethernet network.

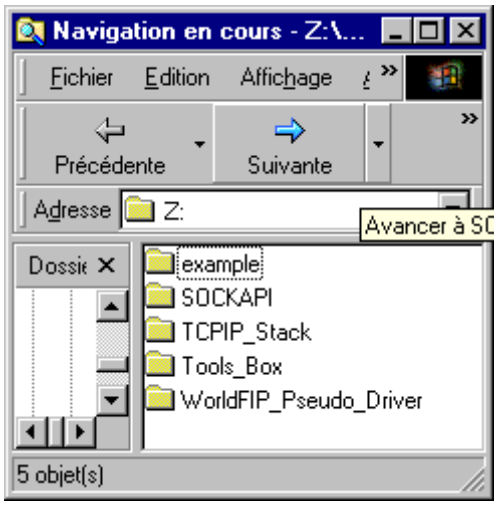
The downside of this operating system independence is a greater effort required on the part of the user, who has to install the software into his own specific operating system and complete some empty functions.

The installation and integration steps will be covered in detail throughout this document.

Chapter 2 General overview of the software

1. SOFTWARE PACKAGE

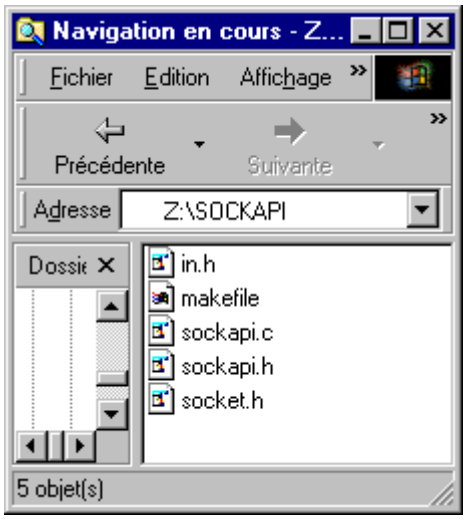
The TCP/IP-WorldFIP Suite package is composed of five source code directories:



- **SOCKAPI**: library that implements the sockets.
- **TCPIP_Stack**: library that implements the TCP/IP Stack.
- **WorldFIP_Pseudo_Driver**: library that implements a pseudo-device driver for a WorldFIP network based on the FIP DEVICE MANAGER R4 library.
- **Tools_Box**: library that implements a toolbox used by the above libraries.
- **example**: example

1.1 SockAPI module

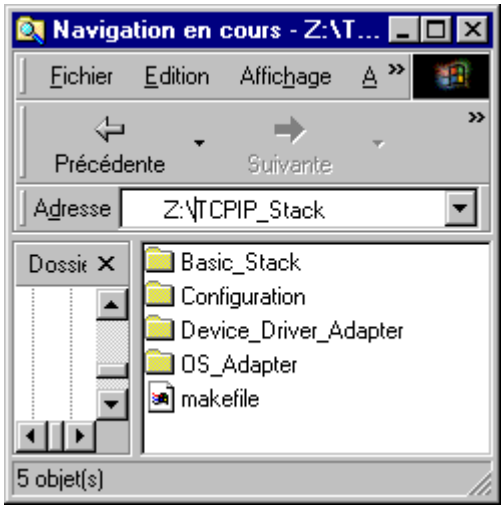
The SockAPI module is composed of the following files:



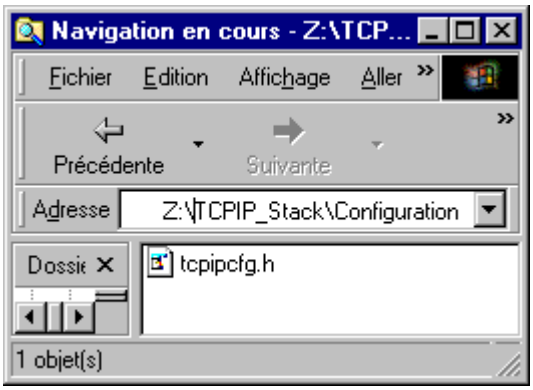
- source files:
  - *sockapi.c*: implements the SockAPI library functions
- header files:
  - *in.h*,
  - *socket.h*,
  - *sockapi.h*.
- miscellaneous files:
  - *makefile*: example of a make-file to build the SockAPI library; this make-file has to be modified by the user (file paths, etc.).

1.2 TCP/IP Stack module

The TCP/IP Stack module is composed of the following directories:

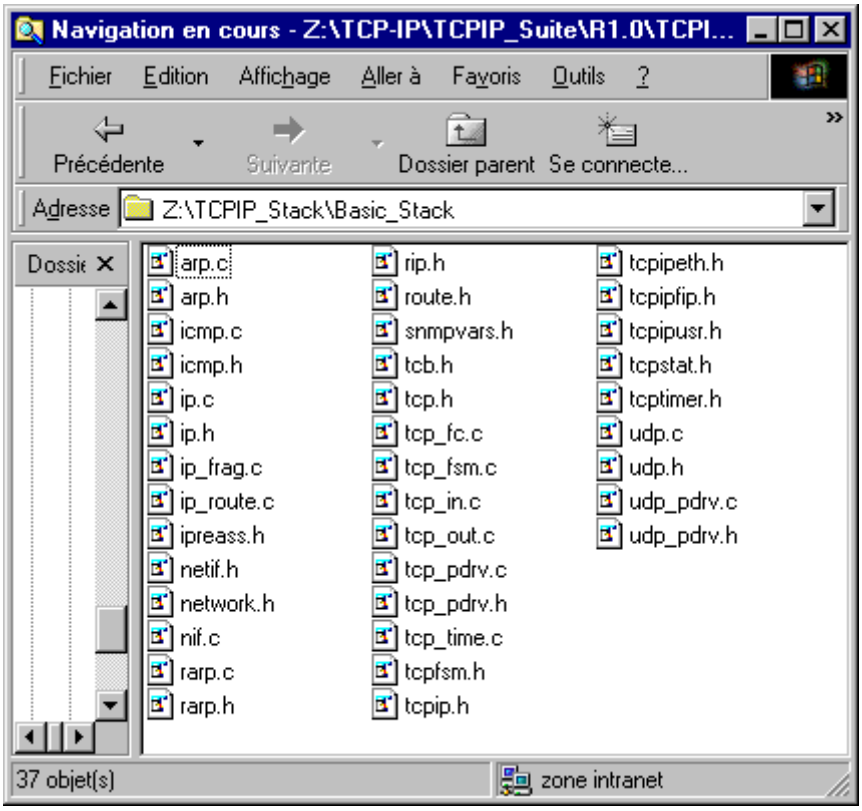


- The **Configuration** directory is composed of the following file:



  - *tcpipcfg.h:* contains the TCP/IP Stack parameters; you should modify this file according to the utilisation context.

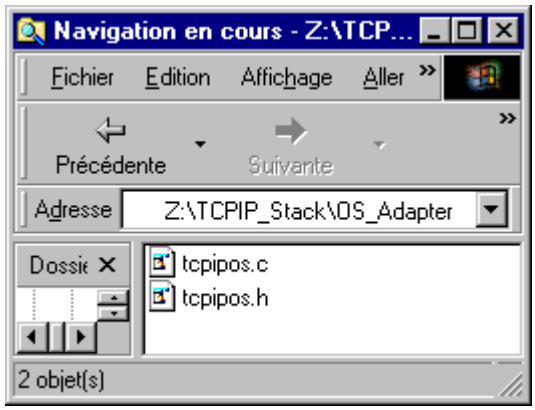
- The **Basic_Stack** directory is composed of the following files:



- source files:
  - *arp.c*: contains the functions of the ARP protocol,
  - *icmp.c:* contains the functions of the ICMP protocol,
  - *ip.c*: contains the general functions of the IP protocol,
  - *ip_frag.c*: contains the fragmentation and assembly functions of the IP protocol,
  - *ip_route.c*: contains the routing functions of the IP protocol,
  - *nif.c*: contains the initialisation functions of the TCP/IP module,
  - *rarp.c*: contains the functions of the RARP protocol,
  - *tcp_fc.c:* contains the control of flow and adaptive retransmission functions of the TCP protocol,
  - *tcp_fsm.c:* contains the state machines functions of the TCP protocol,
  - *tcp_in.c*: contains the input functions of the TCP protocol,
  - *tcp_out.c:* contains the output functions of the TCP protocol,
  - *tcp_pdrv.c*: contains the functions of a TCP pseudo-driver,
  - *tcp_time.c*: contains the time-out management functions of the TCP protocol,
  - *udp.c*: contains the functions of the UDP protocol,
  - *udp_pdrv.c*: contains the functions of a UDP pseudo-driver,

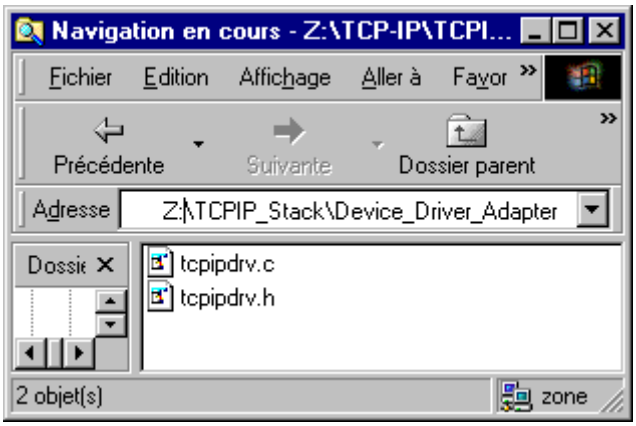
- header files:
  - *arp.h:* contains the type definitions used by the ARP protocol,
  - *icmp.h:* contains the type definitions used by the ICMP protocol,
  - *ip.h*: contains the general type definitions used by the IP protocol,
  - *ipreass.h:* contains the type definitions used by the IP protocol for frame fragmentation/assembly,
  - *netif.h*: contains the type definitions of the Network Interface module,
  - *network.h:* contains the macro transcription of the types (little endian / big endian),
  - *rarp.h*: contains the type definitions used by the RARP protocol,
  - *rip.h:* contains the type definitions used by the RIP protocol,
  - *route.h:* contains the type definitions used by the IP protocol for routing,
  - *snmpvars.h*: contains the type definitions used for statistics management.
  - *tcb.h:* contains the type definitions used by the TCP protocol for the description of the TCB (Transmission Control Block),
  - *tcp.h:* contains the general type definitions used by the TCP protocol,
  - *tcp_pdrv.h*: contains the type definitions used by the TCP pseudo-driver,
  - *tcpfsm.h*: contains the type definitions used by the TCP protocol for the states machine,
  - *tcpip.h:* contains the *include* of all the other h files,
  - *tcpipeth.h*: contains the type definitions used in interfacing with Ethernet,
  - *tcpipfip.h:* contains the type definitions used in interfacing with WorldFIP,
  - *tcpipusr.h*: contains the type definitions common to the TCP/IP and the user,
  - *tcpstat.h:* contains the type definitions used by the TCP protocol for the Status Control definition,
  - *tcptimer.h:* contains the type definitions used by the TCP protocol for time-out management,
  - *udp.h:* contains the type definitions used by the UDP protocol,
  - *udp_pdrv.h*: contains the type definitions used by the UDP pseudo-driver,

- The **OS_Adapter** directory is composed of the following files:



  - *tcpipos.c, tcpipos.h:* adaptation to the operating system; the user has to modify these files according to the utilisation context.

- The **Device_Driver_Adapter** directory is composed of the following files:



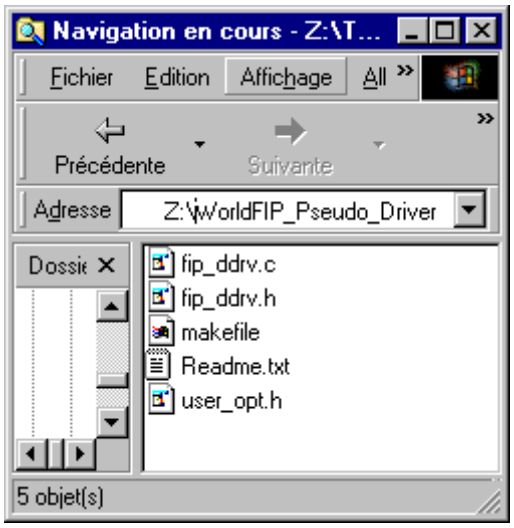
  - *tcpipdrv.c, tcpipdrv.h:* adaptation to a device driver; the user has to modify these files according to the utilisation context.

- miscellaneous files:

  - *makefile*: example of a make-file to build the TCP/IP library; the user has to modify this make-file (file paths, etc.).

1.3 WorldFIP pseudo-device driver module

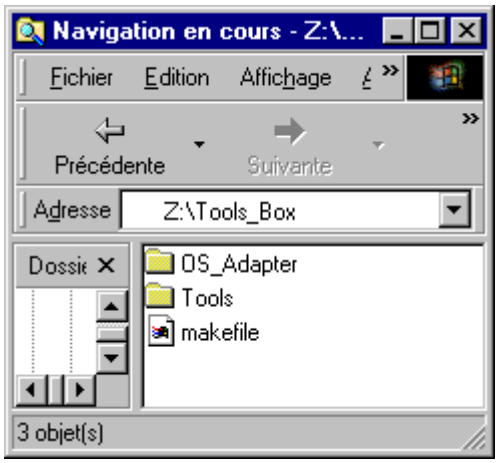
The WorldFIP Pseudo-Device Driver module is composed of the following files:



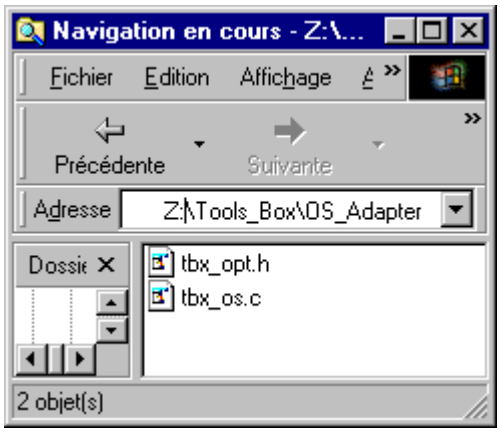
- source files:
  - *fip_ddrv.c*: implements the functions of the WorldFIP pseudo-driver,
- header files:
  - *fip_ddrv.h.*
- miscellaneous files:
  - *user_opt.h:* example of a *user_opt.h*; you can modify the *user_opt.h* file from the FDM package as in this example.
  - *readme.txt:* text file that provides important information on the options to be set in the *tcpipcfg.h* file of the TCP/IP Stack module and the options to be set in the *user_opt.h* file.
  - *makefile*: example of a make-file to build the pseudo-driver library; you should modify this make-file (file paths, etc.).

1.4 Tools Box module

The Tools Box module is composed of the following directories:

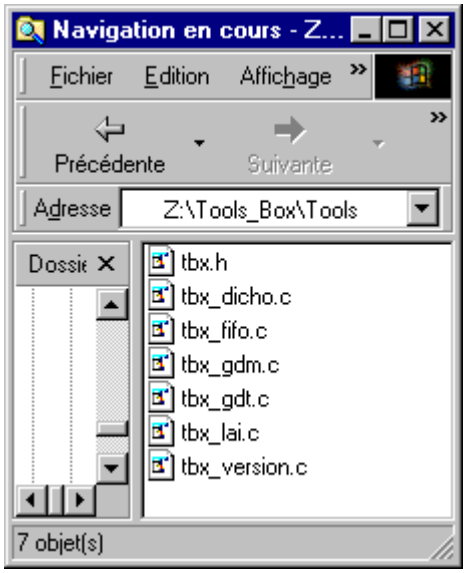


- The **OS_Adapter** directory is composed of the following files:



  - *tbx_os.c:* implements semaphores and critical sections; if the operating system is not pSOS, VxWorks or NT4 then you have to complete this file according to its operating system.
  - *tbx_opt.h:* contains the compilation options of the Tools Box module; you should modify this file according to the utilisation context.

- The **Tools** directory is composed of the following files:



- source files:
  - *tbx_dicho.c:* implements the functions of some sorting algorithms,
  - *tbx_fifo.c:* implements FIFO management functions,
  - *tbx_gdm.c:* implements memory management functions,
  - *tbx_gdt.c*: implements time-out management functions,
  - *tbx_lai.c:* implements inter-module management functions,
  - *tbx_version.c:* gets version number of the Tools Box Software,
- header files:
  - *tbx.h*: contains the definitions of the Tools Box module,
- miscellaneous files:
  - *makefile*: example of a make-file to build the Tools Box library; you should modify this make-file (file paths, etc.).

1.5 Example

This directory contains an example of a pSOS system that shows you how to:

- start and implement the operating system adaptation and the sequencer of the TCP/IP Stack module,
- manage the FIP DEVICE MANAGER messaging mechanism.

The Example directory is composed of the following files:

- *appli.cpp*: contains your application,
- *fipmgt.cpp*:
  - creates an FDM network context,
  - creates the tasks that manage FDM messaging and the IRQ and EOC interrupts of FULLFIP2.
- *root.cpp*: implements the root task.
- *tcpipos.c:* implements the adaptation to pSOS operating system of the TCP/IP Stack module,
- *tcpipmgt.cpp*:
  - configures the structures for the start of the TCP/IP Stack module,
  - creates the tasks that manage the sequencer of the TCP/IP Stack module.

2. SOFTWARE CONFIGURATION

The user has to configure the **TCP/IP Stack** module according to his specific utilisation.

Modify the *TCPIP_Stack/Configuration/tcpipcfg.h* file to select the target machine's operating system and define the stack services which have to be activated, and set the operating parameters of the various active services (for further details see Chapter 3).

3. OPERATING SYSTEM ADAPTATION

The user has to adapt the following modules to his operating system:

- TCPIP_Stack,
- Tools_Box.

The user has to modify the following files:

- *TCPIP_Stack/OS_Adapter/tcpipos.c*,
- *TCPIP_Stack/OS_Adapter/tcpipos.h*,
- *Tools_Box/OS_Adapter/tbx_opt.h*,
- *Tools_Box/OS_Adapter/tbx_os.c*.

For further details see Chapter 4.

4. TCP/IP SEQUENCER

The **TCP/IP Stack** module implements the standard layers (TCP, IP etc.). The operation of this stack module causes messages to transit between the various internal layers (TCP, IP).

The communication between these internal layers is ensured by a set of message queues. Whenever layer A (for example TCP) transmits a message to layer B (for example IP) then layer A writes a message in a shared queue between layer A and layer B; a user function of a Sequencer module will be called. You can request the emptying of the queue to be processed by module B whenever you wish.

The main objective of this mechanism is to enable, as far as possible, the running of the **TCP/IP Stack** module independently from the activation context of an internal layer. The user can choose the time and the context to run the processing code of a destination layer, independently of the moment and the context of the request by a sending layer.

Note

The user has to implement the Sequencer module (see Chapter 5).

5. DEVICE DRIVER ADAPTATION

The access to a WorldFIP network is provided via a pseudo-driver mapped on the ALSTOM FIP DEVICE MANAGER library. In this case you have to set the following options in the *TCPIP_Stack/Configuration/tcpipcfg.h* file:

- **`WORLDFIP_PSEUDO_DRIVER`** must be set to **`TCPIP_YES`**,
- **`FIP_DRV_IND`** must be set to **`TCPIP_YES`**.

If you want to map the TCP/IP Stack module on an Ethernet driver or on another WorldFIP driver, you have to modify the *TCPIP_Stack/Device_Driver_Adapter/tcpipdrv.c* file (for further details see Chapter 6). If you use the pseudo-driver mapped on the ALSTOM FIP DEVICE MANAGER library then you don't have to modify this file.

6. USER INTERFACE

The user can access the TCP/IP communication functions via:

- Socket functions,
- TCP/IP Stack functions.

6.1 Socket functions

The SockAPI module defines a network programming interface which is based on the *socket* paradigm popularised in Berkeley Software Distribution (BSD) from the University of California. A socket defines a bi-directional endpoint for communication between processes (on the same or on different machines).

The Sockets specification is intended to provide a single API with an abstraction of the networking software below it.

The present SockAPI implementation provides an abstraction for the TCP/IP Stack developed by ALSTOM.

The SockAPI library offers service for both connection-oriented (TCP) and connectionless (UDP) protocols. A connection-oriented socket is called a **stream socket** and a connectionless socket is called a **datagram socket**.

There are 2 socket types:

- **blocking sockets**: the operations on these sockets are blocking; i.e. invoking a function which does not return until the associated operation is completed.
- **non blocking sockets**: the operations on these sockets are non blocking.

There are 2 types of services provided by the SockAPI library:

- standard services as defined in Berkeley Software Distribution (BSD),
- non standard services used for launching or stopping the SockAPI library or for error handling.

6.1.1 Standard services:

The following standard services are implemented:

Function	Description
`socket()`	creates a socket,
`bind()`	associates a socket to a local endpoint address,
`listen()`	sets up a connection queue in the case of a stream socket for a Server,
`accept()`	removes the next connection request (from a Client) from the above queue in the case of a stream socket for a Server and creates a new socket; the Server will interact with the Client on the new created socket,
`connect()`	allows a Client to specify the remote endpoint address for a previously created socket; in the case of a stream socket it establishes the connection,
`recv()`	extracts the next message that arrives on the stream socket,
`recvfrom()`	extracts the next message that arrives on the datagram or the stream socket,
`send()`	sends the message to the remote station on a stream socket,
`sendto()`	sends the message to the remote station on a datagram socket,
`closesocket()`	terminates the connection gracefully and removes the socket

6.1.2 Non standard services:

The following non standard services are implemented:

Function	Description
`sockapi_start()`	This function must be the first function called by an application; it initialises the internal library resources.
`sockapi_stop()`	This function must be the last function called by an application, when the application no longer needs the library functions,
`sockapi_get_last_error()`	When any of the standard functions fails, the application can call this function to get the last error,
`sockapi_ioctl()`	This function can set: • the operating mode of a socket: blocking or non blocking, • the Nagle algorithm for the TCP layer, • the buffered delivery mode from the TCP layer.
`sockapi_ip2dot()`	converts an IP address into a dotted decimal string,
`sockapi_dot2ip()`	converts a dotted decimal string into an IP address

6.2 TCP/IP Stack specific functions

The TCP/IP Stack specific functions in the following list are detailed in the following sections:

Function	Description
tcpip_start()	starts the TCP/IP Stack module (further details see Chapter 7, Section 1)
tcpip_stop()	stops the TCP/IP Stack module (further details see Chapter 7, Section 3)
tcpip_echo_rq() **user_echo_reply()**	exchanges an ICMP type ECHO message between two sites (further details see Chapter 9, Section 6)
tcpip_get_subnetmask() **user_mask_reply()**	ICMP functions to send the request for a subnet mask and to receive the subnet mask (further details see Chapter 9, Section 7)
tcpip_conf_tarp()	configures in a static manner (without ARP protocol) the correspondence table between IP addresses and hardware addresses for the various locations (further details see Chapter 9, Section 3)
tcpip_getaddr()	requests the IP address of a local interface (further details see Chapter 9, Section 1)
tcpip_putaddr()	sets the IP address of a local interface (further details see Chapter 9, Section 2)
tcpip_rtadd()	adds an element from the routing table (further details see Chapter 9, Section 4)
tcpip_rtdel()	deletes an element from the routing table (further details see Chapter 9, Section 5)

7. APPLICATION CREATION

7.1 Libraries

A user application for a WorldFIP network must be linked to the following modules (libraries):

- *tcpip.lib*: library implementing the functions of the **TCP/IP Stack** module,
- *sockapi.lib*: library implementing the functions of the **SockAPI** module,
- *tbx.lib*: library implementing the functions of the **Tools Box** module,
- *wfip_ddrv.lib*: library implementing the functions of the **WorldFIP Pseudo-Driver** module mapped on a ALSTOM **FIP DEVICE MANAGER** library.
- *fdm.lib:* library implementing the functions of a **FIP DEVICE MANAGER** module.

Note

The names of the above libraries are only examples. You can change them.

7.2 How to create an application that uses the TCP/IP - WorldFIP Suite package:

To create an application that uses the TCP/IP WorldFIP Suite package you should proceed as follows:

1. Modify the *TCPIP_Stack/Configuration/tcpipcfg.h* file (for details see Chapter 3) to adapt the parameters of the **TCP/IP Stack** module to your context.
2. Modify the *TCPIP_Stack/OS_Adapter/tcpipos.c* file and if necessary the *TCPIP_Stack/OS_Adapter/tcpipos.h* file to adapt the **TCP/IP Stack** module to your operating system (for details see Chapter 4).
3. Modify the *TCPIP_Stack/Device_Driver_Adapter/tcpipdrv.c* file, if you want to map the **TCP/IP Stack** module on an Ethernet driver or on another WorldFIP driver (for details see Chapter 6). This step is not necessary, if you use the **WorldFIP pseudo-driver** module.
4. Implement a **Sequencer** module (for details see Chapter 5).
5. Create the *tcpip.lib* library.
6. Create the *sockapi.lib* library that implements the functions of the **SockAPI** module.
7. If you use the **WorldFIP pseudo-driver** module then:
   - create the *wfip_ddrv.lib* library,
   - create the *fdm.lib* library.
8. Modify the *Tools_Box/OS_Adapter/tbx_opt.h* file and if necessary the *Tools_Box/OS_Adapter/tbx_os.c* file to adapt the **Tools Box** module to your operating system.
9. Create the *tbx.lib* library.
10. Implement a **Sequencer** module for the messaging treated by the FIP DEVICE MANAGER (for details see the FIP DEVICE MANAGER User Reference Manual [ALS 50278**]**).
11. Inside the body of the real-time clock handler call the **`tbx_TimerUpToDate()`** function (see Chapter 10).
12. Start the **TCP/IP Stack** module with the **`tcpip_start()`** function.
13. Start the **SockAPI** module with the **`sockapi_start)`** function.
14. Use the functions of the **SockAPI** module (see Chapter 8).
15. Use the functions of the **TCP/IP Stack** module (see Chapter 9)).
16. If you no longer needs to use the **TCP/IP Suite** functions then stop the **SockAPI** module with the **`sockapi_stop()`** function only after having received all the answers to the requests in progress.
17. Stop the **TCP/IP Stack** module with the **`tcpip_stop()`** function.

Chapter 3 Software configuration

The user has to configure the TCP/IP Stack module according to his specific utilisation.

In the *TCPIP_Stack/Configuration/tcpipcfg.h* file you can:

- select the target machine's operating system,
- define the stack services needing to be activated,
- set the operating parameters of the various active services.

These parameters have been grouped into a single file. They are represented by functional groups, from Figure 3.1 to Figure 3.10.

1. SELECTION OF THE TARGET MACHINE'S OPERATING SYSTEM

The figure below shows you the "defines" that you can use to select the processor type, the operating system and the task architecture.

```
Activating the Processor Type, Operating System and Task Architecture

#define LITTLE_ENDIAN      TCPIP_NO        /* processor type :
                                              TCPI_YES for little endian
                                              TCPI_NO  for big endian */
#define PSOS               TCPIP_YES       /* PSOS OS :
                                              TCPI_YES if you use pSOS
                                              TCPI_NO  if you don't use pSOS */
#define UNIX               TCPIP_NO        /* UNIX OS :
                                              TCPI_YES if you use UNIX
                                              TCPI_NO if you don't use UNIX */
#define DOS                TCPIP_NO        /* DOS OS :
                                              TCPI_YES if you use DOS
                                              TCPI_NO if you don't use DOS */
#define WIN_NT             TCPIP_NO        /* NT OS :
                                              TCPI_YES if you use NT
                                              TCPI_NO if you don't use NT */
#define MONO_TASK          TCPIP_NO        /* task architecture :
                                              TCPI_YES for mono task
                                              TCPI_NO for multi tasks */
```

Figure 3.1 - Selection of the operating system

2. ACTIVATION OF THE TCP/IP STACK'S OPTIONAL SERVICES

Basic services, such as IP, ARP or ICMP are active by default. The user should activate the other services (TCP, UDP, RARP, FRAGMENT and RIP). The following figure contains the list of optional services.

```
                                    Activating optional TCP/IP Stack services

#define TCP                TCPIP_YES       /* TCP module :
                                              TCPI_YES if you use TCP
                                              TCPI_NO  if you don't use TCP */
#define UDP                TCPIP_YES       /* UDP module :
                                              TCPI_YES if you use UDP
                                              TCPI_NO  if you don't use UDP */
#define FRAGMENT           TCPIP_YES       /* IP fragmentation/re-assembly :
                                              TCPI_YES IP have to F/R the frames
                                              TCPI_NO  IP don't have to */
#define RARP               TCPIP_NO        /* RARP module :
                                              TCPI_YES if you use RARP
                                              TCPI_NO if you don't use RARP */
#define RIP                TCPIP_NO        /* RIP protocol implementation :
                                              TCPI_YES if RIP is implemented
                                              TCPI_NO if RIP is not implemented */
```

Figure 3.2 - Service Selection

3. SET THE VALUE OF THE TIMER TICK

```
                                                        Timer Tick Value
#define TIMER_TICK              100             /* measured in milliseconds */
```

Figure 3.3 - Timer Tick Value

4. SELECTION OF THE WORLDFIP PSEUDO DRIVER

```
                                  Selection of the network interface pseudo driver
#define FIP_DRV_IND             TCPIP_YES       /* Reception mode of WorldFIP packets :
                                                   TCPI_YES direct call by the driver
                                                   of the function tcpip_rcvpack()
                                                   TCPI_NO the user must implement a
                                                   function of the driver, and then
                                                   call tcpip_rcvpack() */

#define WORLDFIP_PSEUDO_DRIVER TCPIP_YES        /* use of the ALSTOM WorldFIP pseudo-driver :
                                                           TCPIP_YES if the WorlFIP pseudo
driver
                                                           is used
                                                           TCPIP_NO if the WorlFIP pseudo
```

Figure 3.4 - Selection of the WorldFIP pseudo driver

5. GENERIC NETWORK INTERFACE CONFIGURATION PARAMETERS

```
                                Generic network interface configuration parameters

#define NMAXETH              0     /* max. number of Ethernet interfaces
                                      should be 0 if you don't have Ethernet interface */
#define NMAXFIP              1     /* max. number of WorldFIP interfaces
                                      should be 0 if you don't have WorldFIP interface */

#define NB_OF_ETH_BUF       64     /* number of Ethernet buffers */
#define MAX_ETH_MTU       1500     /* Ethernet Maximum Transmit Unit */
#define SIZE_OF_ETH_BUF   1524     /* size of Ethernet buffer =
                                      MTU + size of extended buffer */

#define NB_OF_FIP_BUF       64     /* number of WorldFIP buffers */
#define MAX_FIP_MTU        254     /* WorldFIP Maximum Transmit Unit =
                                      MAX_FIP_MTU + 2 added by the device driver */
#define SIZE_OF_FIP_BUF    280     /* size of WorldFIP buffer =
                                      MTU + size of extended buffer */

#define NB_OF_LRG_BUF       64     /* number of large buffers */
#define SIZE_OF_LRG_BUF   2048     /* size of large buffers */

#define NI_MAXHWA            6     /* max size of any hardware (physical) net address */

#define NETNLEN             30     /* lenght of network name */

#define NB_OF_MSG_IN_FIFOS 64      /* nb. of elements of the pool used for the
                                      inter-module FIFOS */
```

Figure 3.5 - Generic network interface configuration parameters

6. IP PROTOCOL CONFIGURATION PARAMETERS

```
                                IP protocol configuration parameters

#define IP_ALEN                4   /* IP address length (in bytes) */
#define IP_VERSION             4   /* current version value */
#define IP_MINHLEN             5   /* minimum IP address header length (in longs) */
#define IP_TTL                16   /* initial Time-To-Live value */

#define RT_MAX                50   /* max number of routes */
#define RT_UPDATE            100   /* period of sweep of the routing table to decrement
                                      time-to-live values and dispose of routes that have
                                      expired – measured in timer ticks – used only if
                                      RIP protocol is implemented */

#define GW_DEFAULT {192,9,230,3}   /* default gateway */
#define IP_DEFAULT             1   /* default interface to reach the default gateway */
#define DT_DEFAULT             1   /* distance in hops */

#define FG_UPDATE            100   /* period of check of the fragment lists – measured
                                      in timer ticks */
#define IP_FTTL              600   /* time to live of reassembly of fragments – measured
                                      in timer ticks – if after IP_FTTL ticks we don't
                                      have all the the fragments then send an ICMP
                                      « time exceed » */
```

Figure 3.6 - IP Protocol Configuration Parameters

7. ARP PROTOCOL CONFIGURATION PARAMETERS

```
                                        ARP protocol configuration parameters

#define ARP_TSIZE              50   /* max ARP cache size for each interface */
#define ARP_QSIZE              10   /* ARP queue size for the packets waiting ARP
                                       resolution for an address */

#define ARP_RESEND             10   /* resend a ARP packet if no reply – measured
                                       in timer ticks */
#define RARP_RESEND            15   /* resend a RARP packet if no reply – measured
                                       in timer ticks */

#define ARP_TIMEOUT60*ARP_RESEND    /* time out for removing temporary entries in the
                                       ARP table – must be a multiple of ARP_RESEND */
#define ARP_MAXRETRY            4   /* maximum retransmission for ARP and RARP */
```

Figure 3.7 - ARP Protocol Configuration Parameters

8. ICMP PROTOCOL CONFIGURATION PARAMETERS

```
                                      ICMP protocol configuration parameters

#define IC_RDTTL             300   /* Time-To-Live for redirect routes */
#define IC_MAX_ECHO_PROC       3   /* Maximum number of process using the echo request */
```

Figure 3.8 - ICMP Protocol Configuration Parameters

9. UDP PROTOCOL CONFIGURATION PARAMETERS

```
                                        UDP protocol configuration parameters

#define UPPS           11          /* max number of UDP ports */
#define ULPORT       2050          /* initial UDP local port number allocated by
                                      the stack if the user don't specify it */
```

Figure 3.9 - UDP Protocol Configuration Parameters

10. TCP PROTOCOL CONFIGURATION PARAMETERS

```
                                        TCP protocol configuration parameters

#define MAX_ACT_PORTS              0   /* max. number of active ports (for clients) */
#define MAX_PAS_PORTS              1   /* max. number of passive ports (for servers) */
#define MAX_ACCEPTED_CX_ON_A_PAS_PORT  5  /* max. number of accepted connections for each
                                        passive port */

#define NTCB                 (MAX_ACT_PORTS+MAX_PAS_PORTS*(1+MAX_ACCEPTED_CX_ON_A_PAS_PORT))
                                      /* number of TCB blocks =
                                             number of active ports (for clients)  +
                                             number of passive ports (for servers) +
                                             number of passive ports (for servers)*
                                             number of accepted connections for each
                                             passive port                  */

#define TCPSBS                  2048   /* TCP send buffer sizes    */
#define TCPRBS                  4096   /* TCP receive buffer sizes */

/* Important remark : The size of an TCB descriptor used for each connection is :

                                     size of the structure tcb (see tcb.h)
                                       +
                                     TCPSBS : send buffer size
                                       +
                                     TCPRBS : receive buffer size */

#define TCPSBS                  2048   /* TCP send buffer size */
#define TCPRBS                  4096   /* TCP received buffer size */

#define TLPORT                  1024   /* initial TCP local port number allocated
                                        by the stack if the user don't specify it */
#define TCP_MAXRETRIES            12   /* max retransmissions before giving up */
#define TCP_TIMER_PERIOD           1   /* TCP timer period :
                                         - measured in timer ticks
                                         - must be 100 milliseconds */
#define TCP_MAXPRS               600   /* 1 minute Max Persist Time :
                                        must be a multiple de 100 milliseconds */
#define TCP_TWOMSL              1200   /* waiting time before deleting the record
                                        of a connection :
                                         - must be 2 * Max Segment Lifetime
                                         - set to 2 minutes
                                         - must be a multiple de 100 milliseconds */
#define TCP_MAXRXT               200   /* 20 seconds  Max Rexmt Time :
                                        must be a multiple de 100 milliseconds */
#define TCP_MINRXT                 5   /* 1/2 second Max Rexmt Time :
                                         - must be a multiple de 100 milliseconds */
#define TCP_ACKDELAY               2   /* 1/5 second ACK delay, if TCBF_DELACK :
                                         - must be a multiple de 100 milliseconds */
#define BSDURG              TCPIP_NO   /* urgent pointer interpretation
                                        TCPIP_NO  standard interpretation
                                        TCPIP_YES Berkeley interpretation */
```

Figure 3.10 - TCP Protocol Configuration Parameters

11. ALLOCATED RAM MEMORY

11.1 RAM memory allocated by the TCP/IP Stack module

The dynamic memory size in bytes allocated by the **TCP/IP Stack** module according to the parameters of the *TCPIP_Stack/Configuration/tcpipcfg.h* file is represented as follows:

`NB_OF_ETH_BUF*(SIZE_OF_ETH_BUF + 24)`
+
`NB_OF_FIP_BUF*(SIZE_OF_FIP_BUF + 24)`
+
`NB_OF_LRG_BUF*SIZE_OF_LRG_BUF`
+
`NB_OF_MSG_IN_FIFOS*64`
+
`NMAXETH*(ARP_TSIZE*sizeof(ETH_ARPENTRY)` - see the *arp.h* file
+
`NMAXFIP*(ARP_TSIZE*sizeof(FIP_ARPENTRY)` - see the *arp.h* file
+
`RT_MAX*sizeof(struct route)` - see the *route.h* file
+
(number of TCP open connections) * (size of a TCB block)

Note

The NTCB parameter defines the maximum number of connections for the TCP protocol.

Size of a TCB block (in bytes):

Size of`(struct tcb)` – see the *tcb.h* file
+
`TCPSBS`
+
`TCPRBS`

11.2 RAM memory allocated by the WorldFIP pseudo driver

The dynamic memory size allocated by the **WorldFIP Pseudo-Driver** module for a single WorldFIP interface according to the parameters of the *TCPIP_Stack/Configuration/tcpipcfg.h* file is represented below.

According to these parameters the allocated memory size (in bytes) is as follows:

- For transmission:

  `(NB_OF_FIP_BUF + NB_OF_LRG_BUF)`*(size of an internal descriptor)

  The size of the internal descriptor is about 80 bytes.

- For reception:

  `(ptp_nb_rec_buf + brc_nb_rec_buf)`*(size of an internal descriptor)

  The size of the internal descriptor is about 310 bytes.

  `ptp_nb_rec_buf` is the parameter **`if_number_of_rec_buf1`** of the structure **`FIP_SPECIFIC`** used for the number of reception buffers for a point-to-point address. The user sets this parameter when he calls the **`tcpip_start()`** function.

  `brc_nb_rec_buf` is the parameter **`if_number_of_rec_buf2`** of the structure **`FIP_SPECIFIC`** used for the number of reception buffers for a broadcast address. The user sets this parameter when he calls the **`tcpip_start()`** function.

Chapter 4 Operating system adaptation

The user has to adapt the following modules to his operating system:

- **TCPIP_Stack,**
- **Tools_Box**.

1. ADAPTATION OF THE TCPIP_STACK MODULE

To adapt the TCPIP_Stack module you have to modify the following files:

- *TCPIP_Stack/Configuration/tcpipcfg.h,*
- *TCPIP_Stack/OS_Adapter/tcpipos.c,*
- *TCPIP_Stack/OS_Adapter/tcpipos.h,*

To select the target machine's operating system you have to modify the *TCPIP_Stack/Configuration/tcpipcfg.h* file (see Chapter 3).

In a multi-tasking environment, the **TCP/IP Stack** module performs its various processes in parallel, in the form of threads or specialised tasks. These software elements must be synchronised. The access to the common memory must be protected.

By convention these constraints are resolved by semaphores, with their system-dependent implementations.

For this, the **TCP/IP Stack** module provides in the *TCPIP_Stack/OS_Adapter/tcpipos.c* file, four empty predefined functions that the user has to add to the system semaphore management primitives available on his operating system. The four above-mentioned functions are as follows:

Function	Description
`CREATE_SEMAPHORE()`	creates a semaphore,
`DELETE_SEMAPHORE()`	deletes a semaphore,
`ACQUIRE_SEMAPHORE()`	acquires a semaphore,
`RELEASE_SEMAPHORE()`	releases a semaphore

2. ADAPTATION OF THE TOOLS BOX MODULE

You have to modify the following files to adapt the Tools Box module:

- *Tools_Box/OS_Adapter/tbx_opt.h,*
- *Tools_Box/OS_Adapter/tbx_os.c.*

To select the target machine's operating system you have to modify the *Tools_Box/OS_Adapter/tbx_opt.h* file (see below).

```
                                                Activating the Operating System

#define    _TBX_WITH_FDM                        _TBX_OPTION_YES

#define    _TBX_WITH_WinNTUser                  _TBX_OPTION_NO
#define    _TBX_WITH_WinNTKernel                _TBX_OPTION_NO

#define    _TBX_WITH_PSOS                       _TBX_OPTION_YES
#define    _TBX_WITH_VxWorks                    _TBX_OPTION_NO
#define    _TBX_WITH_OtherOS                    _TBX_OPTION_NO

#define    _TBX_WITH_SEMAPHORE                  _TBX_OPTION_NO
```

Figure 4.1 – Tools Box module: Configuration of the tbx_opt.h file

The file *Tools_Box/OS_Adapter/tbx_os.c* implements the following critical section functions to make non interruptible code sections:

Function	Description
`tbx_OSInitRegion()`	The user must place in this empty procedure the appropriate call of his operating system for the initialisation of the interrupt masking mechanism.
`tbx_OSEnterRegion()`	The user must place in this empty procedure the appropriate call of his operating system for interrupt masking.
`tbx_OSLeaveRegion()`	The user must place in this empty procedure the appropriate call of his operating system for interrupt unmasking.
`tbx_OSDeleteRegion()`	The user must place in this empty procedure the appropriate call of his operating system to release the various resources used for the interrupt masking mechanism.

Table 4.1 – Tools Box module: Configuration of the tbx_os.c file

Notes

1. For a certain number of operating systems (pSOS, VxWorks, etc.) these functions are completed. You can use their default values or modify them according to your needs.
2. If the **WorldFIP Pseudo Driver** module is used (so **FIP DEVICE MANAGER** is used), you should set the option **`_TBX_WITH_FDM`** to **`_TBX_OPTION_YES`**. In this case the functions that implement the critical section are those implemented in **FIP DEVICE MANAGER**.

Chapter 5 TCP/IP Sequencer

1. MODULE DESCRIPTION

The **TCP/IP Stack** module implements the standard layers (TCP, IP, etc.). The module operation causes messages to transit between the various internal layers (TCP, IP).

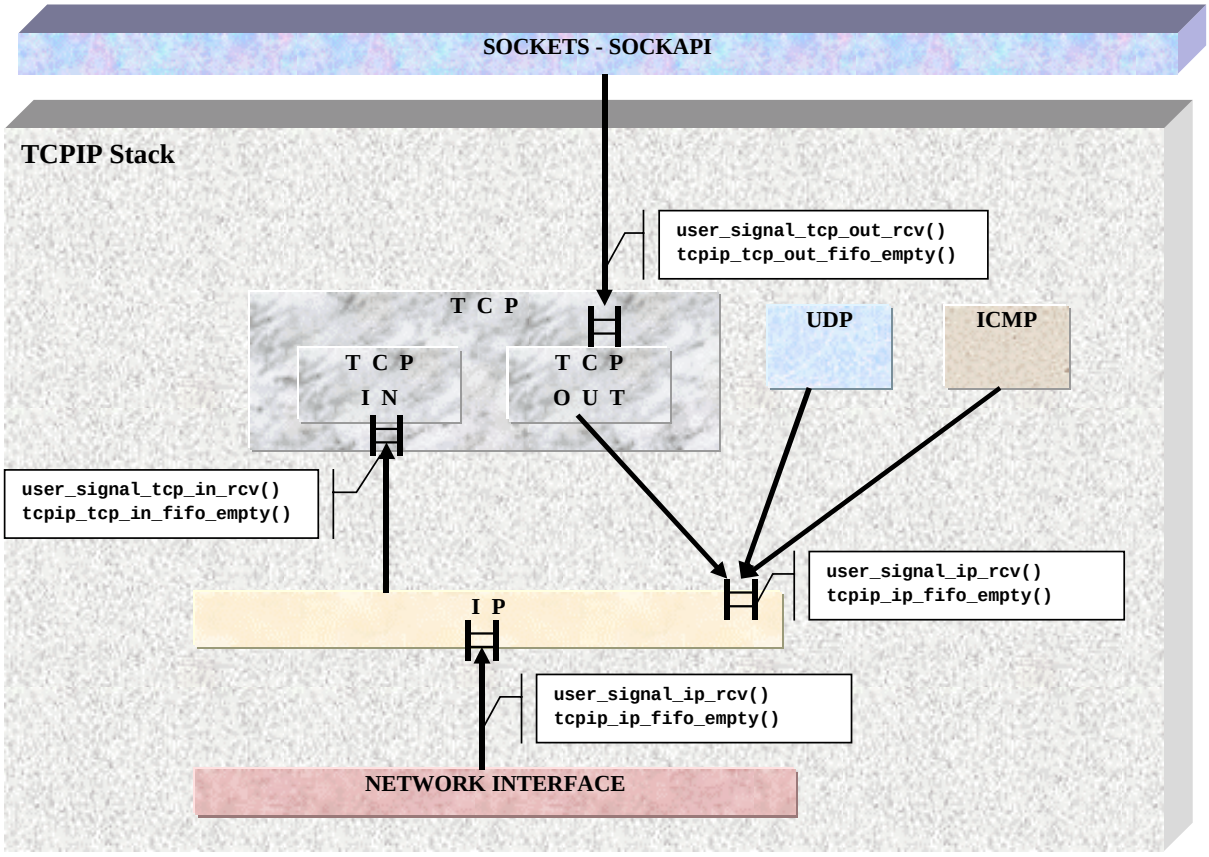


Figure 5.1 – TCP/IP Stack module Sequencer

The communication between these internal layers is ensured by a set of message queues. Whenever a layer A (for example TCP) transmits a message to layer B (for example IP), then layer A writes a message in a shared queue between layer A and layer B, and a user function of a **Sequencer** module will be called. You can request the emptying of the queue to be processed by module B whenever you wish.

From the system point of view the following three activities are processed in parallel:

- application tasks,
- device drivers,
- Sequencer module tasks.

The main objective of the Sequencer is to enable, as far as possible, the running of the **TCP/IP Stack** module, independently from the activation context of an internal layer by application tasks or device drivers. Independently from the moment and the context of the sending layer request, the user can choose the moment and the context to run the processing code of a destination layer.

Each time a message is written in a message queue the **TCP/IP Stack** activation manager is warned of the fact that one of the queues is not empty, when he calls the **`user_signal_xxx_rcv()`** function reserved for this purpose. The user must write this function.

The following functions empty the message queues:

Function	Description
`tcpip_ip_fifo_empty()`	empties the IP queues; these queues are written by the UDP, ICMP, TCP_OUT layers and the Network Interface.
`tcpip_tcp_out_fifo_empty()`	empties the TCP_OUT queue; this queue is written by the SOCKAPI module each time a message is sent.
`tcpip_tcp_in_fifo_empty()`	empties the TCP_IN queue; this queue is written by the IP module

Notes

1. The addresses of the user functions **`user_signal_ip_rcv(), user_signal_tcp_in_rcv(),user_signal_tcp_out_rcv()`** are transmitted to the **TCP/IP Stack** module at the time of the call of the **`tcpip_start()`** function.
2. If you have a multi-task operating system, then inside the body of the **`user_signal_xxx_rcv()`** functions you have to activate a task that will empty the message queue; never call the **`tcpip_xxx_fifo_empty()`** function inside the body of the **`user_signal_xxx_rcv()`** function because its execution context should be different.

Syntax

```
#include "tcpip.h"

void  tcpip_tcp_in_fifo_empty()
void  tcpip_tcp_out_fifo_empty()
void  tcpip_ip_fifo_empty()
```

2. EXAMPLES

On the CD package the file *example/tcpipmgt.cpp* gives you an example of how to implement a Sequencer module on a pSOS operating system.

2.1 Example of a sequencer in a multi-task context

This example shows you how to implement a Sequencer module in a multi-task context.

Execution contexts of the internal layers:

- The IP layer is executed in the context of Task1.
- The TCP_IN layer is executed in the context of Task2.
- The TCP_OUT layer is executed in the context of Task3.

```
#include "tcpip.h"

SEMAPHORE_TYPE        sem1;
SEMAPHORE_TYPE        sem2;
SEMAPHORE_TYPE        sem3;

main_task()
{
.....
        /* create the semaphores */

        CREATE_SEMAPHORE(1, &sem1);
        CREATE_SEMAPHORE(1, &sem2);
        CREATE_SEMAPHORE(1, &sem3);
.....

        /* create the tasks Task1, Task2 et Task3 and then start these tasks */

.....
}


user_signal_ip_rcv()
{
        RELEASE_SEMAPHORE(sem1);      /* wake up the task Task1 */
}


user_signal_tcp_in_rcv()
{
        RELEASE_SEMAPHORE(sem2);      /* wake up the task Task2 */
}


user_signal_tcp_out_rcv()
{
        RELEASE_SEMAPHORE(sem3);      /* wake up the task Task3 */
}
```

```
Task1()
{
        /* task initialisations */
        ..........................................................
        while(1){
                ACQUIRE_SEMAPHORE(sem1);      /* suspend the task */
                tcpip_ip_fifo_empty();
        }
}


Task2()
{
        /* task initialisations */
        ..........................................................
        while(1){
                ACQUIRE_SEMAPHORE(sem2);      /* suspend the task */
                tcpip_tcp_in_fifo_empty();
        }
}


Task3()
{
        /* task initialisations */
        ..........................................................
        while(1){
                ACQUIRE_SEMAPHORE(sem3);      /* suspend the task */
                tcpip_tcp_out_fifo_empty();
        }
}
```

Figure 5.2 – Implementation of a sequencer in a multi-task context

2.2 Example of a sequencer in a mono-task context

This example shows you how to implement a sequencer module in a mono-task context.

A timer calls periodically (10 ms) the **`handler_timer()`** function.

You should complete the following functions of the *Tools_Box/OS_Adapter/tbx_os.c* file which make non interruptible code sections:

Function	User action
`tbx_OSInitRegion()`	The user must place in this empty procedure the appropriate call to his operating system for the initialisation of the mechanism for masking the timer interrupts.
`tbx_OSEnterRegion()`	The user must place in this empty procedure the appropriate call to his operating system for the masking of the timer interrupts.
`tbx_OSLeaveRegion()`	The user must place in this empty procedure the appropriate call to his operating system for the unmasking of the timer interrupts.
`tbx_OSDeleteRegion()`	The user must place in this empty procedure the appropriate call to his operating system to release the various resources used for the mechanism of masking the timer interrupts.

Table 5.1 – Sequencer in a mono-task context (functions)

```
.....

static int activer = 1;

.....

.....


        /* handler_timer : activation by a timer each 10 msec. */

handler_timer()
{
        static int active_timer = 0;

        if( active_timer%10 == 0 ){
                tbx_TimerUpToDate();      /* each 100 msec */
        }
        else{
                if(activer == 1)
                   activer_modules();
        }
        active_timer++;
}

static void activer_modules()
{
        activer = 0;

                /* empty the IP message queue */
        tcpip_ip_fifo_empty();

                /* empty the TCP_IN message queue */
        tcpip_tcp_in_fifo_empty();

                /* empty the TCP_OUT message queue */
        tcpip_tcp_out_fifo_empty();

        activer = 1;
}


static void user_signal_tcp_in_rcv(void *msg)
{

}

static void user_signal_tcp_out_rcv(void *msg)
{

}

static void user_signal_ip_rcv(void *msg)
{

}
```

Figure 5.3 – Implementation of a sequencer for a mono-task context

Chapter 6 Device driver adaptation

1. OVERVIEW

The access to a WorldFIP network is provided via a pseudo-driver mapped on the ALSTOM FIP DEVICE MANAGER library. In this case you have to set the following options in the *TCPIP_Stack/Configuration/tcpipcfg.h* file:

- **`WORLDFIP_PSEUDO_DRIVER`** must be set to **`TCPIP_YES`**.
- **`FIP_DRV_IND`** must be set to **`TCPIP_YES`**.

Modify the *TCPIP_Stack/Device_Driver_Adapter/tcpipdrv.c* file, if you want to map the **TCP/IP Stack** module on an Ethernet driver or on another WorldFIP driver. If you use the pseudo-driver mapped on ALSTOM FIP DEVICE MANAGER library then you don't have to modify this file.

The TCP/IP Stack's Network Interface layer requires an interface with a pseudo device driver in the form of the three following functions and in compliance with the diagram shown below.

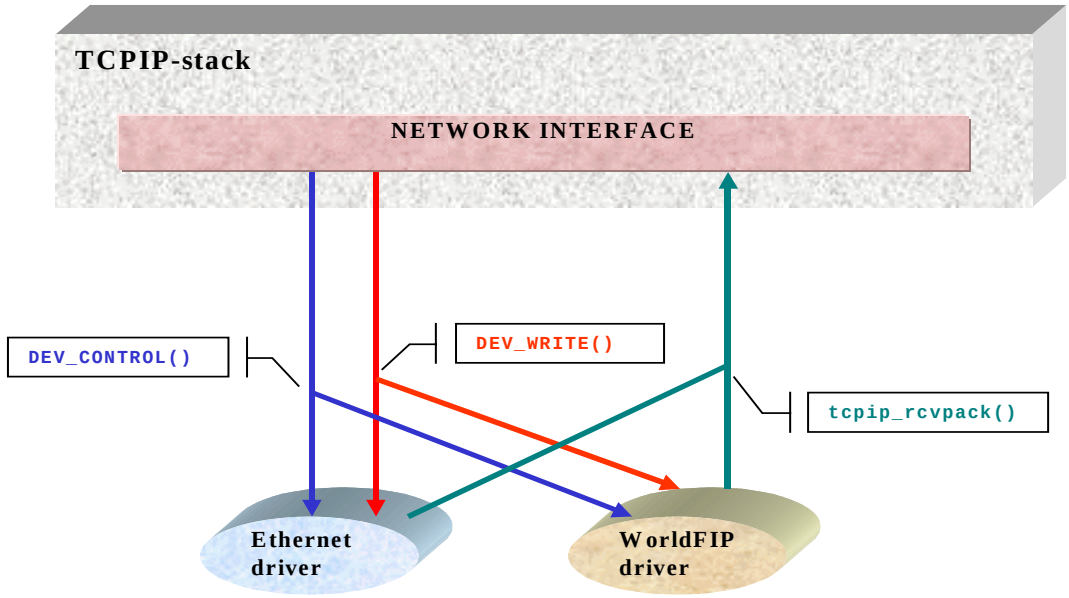


Figure 6.1 - Device Driver Adaptation

- **`DEV_CONTROL()`** handles device driver control operations. Its role is three-fold; it:
  - initialises the resources and elements required for driver messaging (run once only at TCP/IP Stack startup),
  - requests freeing of all pseudo driver resources (run once only at complete stop of the TCP/IP Stack),
  - requests freeing of a pseudo driver receive resource (run after each incoming message).
- **`DEV_WRITE()`** for issuing messages to the network driver.
- **`tcpip_rcvpack()`** the pseudo driver needs to call this function to inform of, and forward, each received message to the TCP/IP stack. The address of this function is transmitted to the device driver by the Network Interface layer at TCP/IP Stack start-up.

2. FUNCTIONS

2.1 DEV_CONTROL(): control operations

2.1.1 Description

The TCP/IP Stack module carries out the following control operations on a device driver:

- initialises the resources and elements required for driver messaging (run only once at TCP/IP Stack start-up),
- requests freeing of all pseudo driver resources (run only once at complete stop of the TCP/IP Stack),
- requests freeing of a pseudo driver receive resource (run after each incoming message); this operation is available only in the case of a WorldFIP interface, when the option in the **`FIP_DRV_IND`** *TCPIP_Stack/Configuration/tcpipcfg.h* file is set to **`TCPIP_YES`**.

The TCP/IP Stack carries out a call of the **`DEV_CONTROL()`** function which is implemented in the form of an empty function in the *TCPIP_Stack/Device_Driver_Adapter/tcpipdrv.c* file.

The user writes the body of this function and carries out the adequate call of its operating system (for example **`ioctl()`**).

Function syntax:

```
#include "tcpip.h"

DEV_CONTROL(IF_DEV_TYPE dev,  int ctrl,  void *args)
```

2.1.2 Inputs

`dev`: number of the device driver; if necessary you have to set **`IF_DEV_TYPE`** in the *TCPIP_Stack/Device_Driver_Adapter/tcpipdrv.h* file; the user gets this number when the device driver is being opened.

`ctrl`: command number:

`INIT_DEV`: initialises the resources and elements required for driver messaging (run only once at TCP/IP Stack start-up),

`STOP_DEV`: requests freeing of all pseudo driver resources (run only once at complete stop of the TCP/IP Stack),

`FREE_MEM_DEV`: requests freeing of a pseudo driver receipt resource (run after each incoming message); this operation is available only in the case of a WorldFIP interface, when the option **`FIP_DRV_IND`** in the *TCPIP_Stack/Configuration/tcpipcfg.h* is set to **`TCPIP_YES`**.

args: pointer to a structure that depends of the value of the **ctrl** parameter:

IF **ctrl = INIT_DEV** THEN

IF driver WorldFIP THEN

args is a pointer to the structure **FIP_MSG_INIT** defined below

ELSE driver Ethernet THEN

args is a pointer to the structure **ETH_MSG_INIT** defined below.

ELSE IF **ctrl = STOP_DEV** THEN

args = 0 (no arguments)

ELSE IF **ctrl = FREE_MEM_DEV** THEN

args = pointer to the reception buffer to be freed

Note

In the case of an Ethernet/WorldFIP router, the interface type depends on the number of the device driver **dev**. You can use global variables for the open devices and then perform processing according to the number of devices whose parameters are set in the **DEV_CONTROL()** function (see the example in Subsection 2.4.2)

2.1.2.1 Description of the FIP_MSG_INIT structure

```
typedef struct {
        unsigned short      protocol;
        unsigned short      if_number;
        TBX_POOL            *fip_pool;
        unsigned short      nb_of_lsaps;
        LSAP_DESC           **lsap_desc_table;
        unsigned short      msg_type;
        unsigned short      channel_nr;
        unsigned short      identifier;
        void                *fdm_ref;
        void                (*indication)(unsigned short, void *, unsigned);
        void                (*confirmation)();
        void                (*addr_user_signal_fip_fatal_error)(unsigned long, int);
        void                (*addr_user_signal_fip_warning)(unsigned long, int)
        TBX_MEMORY_RN       *memory;
        unsigned long       if_report;
}FIP_MSG_INIT;
```

Figure 6.2 – FIP_MSG_INIT structure

Software element	Description
protocol	upper protocol mapped on the driver (for instance only TCP/IP): = 1 → protocol TCPIP
if_number	interface number transmitted by the TCP/IP Stack module; this number has to be memorised by the device driver, because the device driver will use this number as a parameter when it will call the function **tcpip_rcvpack().**
fip_pool	not used
nb_of_lsaps	number of LSAPS (generally 2 LSAPS: one for the point-to-point address and one for the broadcast address).
lsap_desc_table	pointer to an array of **nb_of_lsaps** pointers to structures of the **LSAP_DESC** type defined below.
msg_type	messaging type: = 0 in the case of aperiodical messaging, = 1 in the case of periodical messaging.
channel_nr	channel number: = 0 in the case of aperiodical messaging, = 1 to 8 in the case of periodical messaging
identifier	identifier of the produced variable in the case of periodical messaging; should be set to 0 in the case of aperiodical messaging.
fdm_ref	pointer that is cast to a structure of the **FDM_REF** type (defined in the User Manual of FIP DEVICE MANAGER (see [ALS 50278]).
indication	address of a callback function that should be called whenever a message packet is received; this field is set by the TCP/IP stack module as the address of the **tcpip_rcvpack()** function.
confirmation	address of a callback function that should be called whenever a message is sent; actually this field is not used and should be set to 0.
addr_user_signal_fip_fatal_error	address of a callback function that should be called whenever the driver detects a communication error.
addr_user_signal_fip_warning	address of a callback function that should be called whenever the driver detects a communication warning.
memory	pointer to a structure of the **TBX_MEMORY_RN** type (defined in the file *Tools_Box/Tools/tbx.h*) that allows you to define a memory region (if the operating system permits this) to allocate all the resources for this interface; if the operating system doesn't allow you to define a memory region this parameter should be set to 0.
if_report	return value of interface initialisation; this field is set by the device driver.

Table 6.1 – Elements of the FIP_MSG_INIT structure

2.1.2.2 Description of the LSAP_DESC structure

```
typedef struct {
        unsigned char       lsap_no[FIP_ADDLEN];
        unsigned short      nb_of_rec_buf;
}LSAP_DESC;

#define FIP_ADDLEN 3
```

`lsap_no:` array of 3 bytes that contains the LSAP number.

`nb_of_rec_buf:` number of buffers that should be allocated for the reception of message packets for this LSAP number.

2.1.2.3 Description of the ETH_MSG_INIT structure

```
typedef struct {
        unsigned short      if_number;
        ETH_SPECIFIC        *eth_specific;
        void                (*indication)(unsigned short, void *, unsigned);
        TBX_MEMORY_RN       *memory;
        unsigned long       if_report;
}ETH_MSG_INIT;
```

`if_number:` interface number transmitted by the **TCP/IP Stack** module; this number has to be memorised by the device driver, because the device driver will use this number as a parameter when it calls the **`tcpip_rcvpack()`** function.

`eth_specific:` structure of the **`ETH_SPECIFIC`** type (see Chapter 7).

`indication`: address of a callback function that should be called whenever a message packet is received; this field is set by the TCP/IP Stack module as the address of the function **`tcpip_rcvpack().`**

`memory`: pointer to a structure of the **`TBX_MEMORY_RN`** type (defined in the *Tools_Box/Tools/tbx.h* file) that allows you to define a memory region (if the operating system permits this) where will be allocated all the resources for this interface; if the operating system doesn't allow you to define a memory region then this parameter should be set to 0.

`if_report`: return value of interface initialisation; this field is set by the device driver.

Note

The user has set most of the above parameters when he calls the **`tcpip_start()`** function.

2.1.3 Outputs

0 → OK

-1 → error.

2.2 DEV_WRITE(): sending packets

2.2.1 Description

The TCP/IP Stack module calls the function to send message packets.

`DEV_WRITE()` is implemented in the form of an empty function in the *TPIP_Stack/Device_Driver_Adapter/tcpipdrv.c* file.

The user writes the body of this function and carries out the appropriate call to its operating system (for example **`write()`**).

Function syntax:

```
#include "tcpip.h"

DEV_WRITE (IF_DEV_TYPE dev,  void *xmit_buff, unsigned nbytes).
```

2.2.2 Inputs

`dev`: number of the device driver; if necessary the user has to set **`IF_DEV_TYPE`** in the *TCPIP_Stack/Device_Driver_Adapter/tcpipdrv.h* file; the user gets this number when the device driver is opened.

`xmit_buff`: pointer to a structure whose type depends on the interface type: WorldFIP or Ethernet.

IF driver WorldFIP THEN.

`xmit_buff` is a pointer to a **`FIP_PACK`** structure defined below.

ELSE IF driver Ethernet THEN

`xmit_buff` is a pointer to a structure **`FIP_PACK`** defined below.

`nbytes`: size of the message packet to be sent.

Notes

1. In the case of an Ethernet/WorldFIP router, the type of the interface will be found according to the number of the device driver **`dev`**. You can use global variables for the open devices and then perform processing according to the number of devices whose parameters are set in the **`DEV_CONTROL()`** function (see the example inSubsection 2.1).
2. The device driver must free the **`xmit_buff`** after the sending operation.

Description of the `FIP_PACK` and `ETH_PACK` structures

```
            typedef struct {
               IPaddr              nexthop;
               unsigned short      if_no;
               unsigned short      pack_len;
               unsigned short      reserved;
               STUFFING            stuffing;
               FIP_PACKET          fip_packet;
            }FIP_PACK;


typedef struct {
        Ipaddr              nexthop;
        unsigned short      if_no;
        unsigned short      pack_len;
        unsigned short      reserved;
        ETH_PACKET          eth_packet;
}ETH_PACK;
```

`nexthop` IP address of the next router

`if_no` interface number

`pack_len` packet length (**`FIP_PACK`** or **`ETH_PACK`**)

`reserved` reserved

`stuffing` array of 6 bytes

`fip_packet` structure of the **`FIP_PACKET`** type defined below

`eth_packet` structure of the **`ETH_PACKET`** type defined below

```
Typedef struct{
        FIP_HEADER          fip_header;
        unsigned char       data[FIP_DLEN];
}FIP_PACKET;
```

`fip_header`: structure of the **`FIP_HEADER`** type defined below,

`data`: array of 256 bytes maximum that contains a WorldFIP message.

```
            typedef struct{
               ETH_HEADER          eth_header;
               unsigned char       data[EP_DLEN];
            }ETH_PACKET;
```

`eth_header`: structure of type **`ETH_HEADER`** defined below.

`data`: array of 256 bytes maximum that contains an Ethernet message.

```
typedef struct {
        FIP_ADDR            dst;
        FIP_ADDR            src;
        unsigned short      type;
}FIP_HEADER;
```

dst: hardware destination address (3 bytes),

src: hardware source address (3 bytes),

type: packet type: IP, ARP or RARP.

```
typedef struct {
        ETH_ADDR            dst;
        ETH_ADDR            src;
        unsigned short      type;
}ETH_HEADER;
```

dst: hardware destination address (6 bytes),

src: hardware source address (6 bytes),

type: packet type: IP, ARP or RARP.

2.2.3 Outputs

0 → OK

-1 → error.

2.3 TCPIP_RCVPACK(): receiving packets

2.3.1 Description

To make the system more effective for the reception of message packets, the device driver has to call the **TCPIP_stack** callback function **`tcpip_rcvpack()`** whose address was communicated to it at the time of its initialisation.

This makes it possible to avoid the use of a system call of the **`read()`** type, which is less effective taking into account the asynchronous nature of the reception.

If the driver has not this possibility to carry out the direct call of the **`tcpip_rcvpack()`** function (for example if one uses an already existing device driver which accepts operations of the **`read()`** type), then the user must periodically call the **`read()`** function to this driver, format the received message and call the function **`tcpip_rcvpack()`**.

In the case of the WorldFIP device drivers there are two mechanisms for the allocation of the reception buffers according to the **`FIP_DRV_IND`** option in the *TCPIP_Stack/Configuration/tcpipcfg.h* file:

- **`FIP_DRV_IND`** is set to **`TCPIP_YES`** (as in the case of **ALSTOM WorldFIP pseudo driver**) → the device driver:
  - allocates the buffer for the received packet from its own internal resources pool,
  - writes the received packet inside this buffer,
  - calls the function **`tcpip_rcvpack()`** with the parameter **`rcv_buff`** as a pointer to this buffer.

  When the **TCPIP_stack** module has finished processing the received packet, it will call the function **`DEV_CONTROL()`** with the parameter **`arg`** set to **`FREE_MEM_DEV`** to tell the device driver to free the buffer.

- **`FIP_DRV_IND`** is set to **`TCPIP_NO`** → the user has to write a new function inside the *TPIP_Stack/Device_Driver_Adapter/tcpipdrv.c* file; this function (we name it **`fip_rcv_pack()`** must be called periodically, for example from the real time clock handler. The **`fip_rcv_pack()`** function has to**:**
  - call the device driver **`read()`** function,
  - allocate a buffer for the received packet from the **TCPIP_stack** internal pool created for the WorldFIP interfaces. This pool is created inside the file *TCPIP_Stack/Basic_Stack/nif.c* and the reference to this pool is the global variable **`Net.fip_pool`**. You can use the following function for this allocation:
    - **`FIP_PACK  *fp;`**
    - **`fp = (FIP_PACK *)tbx_AllocateBufferInPool(Net.fip_pool)`**;
  - copy the received packet inside this buffer,
  - call the function **`tcpip_rcvpack()`** with the parameter **rcv_buff** as a pointer to this buffer.

  When the TCP/IP Stack module has finished processing the received packet it will directly free this buffer.

In the case of the **Ethernet device drivers** the user has to write a new function inside the file *TPIP_Stack/Device_Driver_Adapter/tcpipdrv.c***;** this function (we name it **`eth_rcv_pack()`**, must be called periodically, for example from the real time clock handler. The function **`eth_rcv_pack()`** has to:

- create the device driver **`read()`** function,
- allocate a buffer for the received packet from the **TCP/IP stack** internal pool created for the Ethernet interfaces. This pool is created inside the file *TCPIP_Stack/Basic_Stack/nif.c* and the reference to this pool is the global variable **`Net.eth_pool`**.
- You can use the following function for this allocation:
  - **`ETH_PACK *ep;`**
  - **`ep = (FIP_PACK *)tbx_AllocateBufferInPool(Net.eth_pool);`**
- copy the received packet inside this buffer
- call the **`tcpip_rcvpack()`** function with the **rcv_buff** parameter as a pointer to this buffer.

When the **TCP/IP Stack** module has finished processing the received packet it will directly free this buffer.

Function syntax:

```
#include "tcpip.h"

void tcpip_rcvpack(unsigned short if_number, void *rcv_buff, unsigned nbytes)
```

2.3.2 Inputs

`if_number`: interface number; this number is set as a parameter (field **`if_number`** of the structure **`FIP_MSG_INIT`** or **`ETH_MSG_INIT`**) at initialisation of the device driver by the **`DEV_CONTROL`** function.

`rcv_buff`: pointer to a structure whose type depends on the interface type: WorldFIP or Ethernet

IF driver WorldFIP THEN

`rcv_buff` is a pointer to a structure **`FIP_PACK`** defined in the function.

ELSE IF driver Ethernet THEN

rcv_buff is a pointer to a structure **`ETH_PACK`** defined in the **`DEV_WRITE()`** function.

`nbytes`: size of the received packet (**`FIP_PACK`** or **`ETH_PACK`**).

2.3.3 Outputs

None

2.4 Examples

2.4.1 Interfacing an Ethernet driver

This example shows how to interface the **TCP/IP Stack** with an Ethernet device driver (NIF device driver on a Sun station).

This device driver uses a structure of the **`ether_packet`** type (defined below as a parameter). You have to set this structure in the **`DEV_WRITE()`** function.

You have to create the entire **`eth_rcv_pack()`** function and call it periodically.

```
#include "tcpip.h"

typedef struct
{
   ether_addr          dest;
   ether_addr          src;
   unsigned char       type[2];          /* in network byte order! */
   unsigned short      pktlen;           /* length of pktbuf ONLY */
   char                *pktbuf;
}ether_packet;

typedef union
{
   unsigned char       bytes[6];         /* byte order safe initialization
*/
   unsigned short      shorts[3];        /* force 2-byte alignment */
}ether_addr;

unsigned short    eth_if_no;

void              eth_rcv_pack();
void              (*tcpip_rcvpack)(unsigned short, void *, unsigned);
……..
…

/*-----------------------------------------------
 *  DEV_CONTROL : input/output control operations on a device
 -----------------------------------------------
 /

DEV_CONTROL(IF_DEV_TYPE dev, int ctrl, void args)
{
   ETH_MSG_INIT        *eth_msg_init;

      eth_msg_init = (ETH_MSG_INIT *)args;

      switch(ctrl){
```

```
        case INIT_DEV :
            eth_if_no = eth_msg_init->if_number;
            tcpip_rcvpack = eth_msg_init->indication;
            break;

        case STOP_DEV :
            break;

        case FREE_MEM_DEV :
            tbx_FreeBufferInPool((TBX_USER_BUFFER *)args);
            break;

        default :
            break;
      }
  return(0);
}

/*-----------------------------------------------
 *  DEV_WRITE : write operations on a device
*-----------------------------------------------
 */

DEV_WRITE(IF_DEV_TYPE dev, void *xmit_buff, unsigned nbytes)
{
    ether_packet        eth_pack_wr;
    unsigned char       *eth_pack;
    ETH_PACK            *ep;
    struct ip           *pip;
    unsigned short      type;
    unsigned short      hight_type;
    unsigned short      low_type;
    unsigned            i;
    int                 rv;

    ep = (ETH_PACK *)xmit_buff;
    pip = (struct ip *)ep->eth_packet.data;

                                    /* Ethernet destination address */
    for(i = 0; i < 6; i++)
            eth_pack_wr.dest.bytes[i] = ep->eth_packet.eth_header.dst[i];

                                    /* Ethernet source address */
     for(i = 0; i < 6; i++)
            eth_pack_wr.src.bytes[i] = ep->eth_packet.eth_header.src[i];

                                    /* protocol type  */
    type = ep->eth_packet.eth_header.type;
```

```
    hight_type = (type >> 8);
    low_type = (type & 0x00ff);

    eth_pack_wr.type[0] = (unsigned char)hight_type;
    eth_pack_wr.type[1] = (unsigned char)low_type;

                                        /* extended header */
    eth_pack_wr.pktlen = nbytes - ep_hlen;
    eth_pack_wr.pktbuf = (char *)ep->eth_packet.data;

                /* write the packet on the NIT device */

    rv = ether_write(dev, &eth_pack_wr);

              /* free the user buffer */

    tbx_FreeBufferInPool((TBX_USER_BUFFER *)xmit_buff);

    return(rv);
}


/*****************************************************************************
 *  eth_rcv_pack : read the NIT driver (ETHERNET)
*                    this function must be called periodically by the user
 *****************************************************************************
 */

void eth_rcv_pack()
{
    ETH_PACK      *ep;
    struct ip     *pip;
    unsigned short  type;
    unsigned short  hight_type;
    unsigned short  low_type;
    ether_packet   eth_rcv_pk;
    int            eth_pack_len;
    int            i;

    ep = (ETH_PACK *)tbx_AllocateBufferInPool(Net.eth_pool);
    if(ep == 0){
      fprintf(fd,"\n ERROR : Pool eth_pool \n");
      return;
    }

    eth_rcv_pk.pktlen = SIZE_OF_ETH_REC_BUF;
    eth_rcv_pk.pktbuf = (char *)ep->eth_packet.data;

    eth_pack_len = ether_read(eth_dev_fd, &eth_rcv_pk);
    if( eth_pack_len < 20 ){
```

```
                tbx_FreeBufferInPool((TBX_USER_BUFFER *)ep);
                if( errno != EAGAIN )
                  fprintf(fd,"\n ERROR read Ethernet Driver = %d\n",
                                                        eth_pack_len);
                return;
            }

            blkcopy(ep->eth_packet.eth_header.dst,
                              (unsigned char *)eth_rcv_pk.dest.bytes, 6);
            blkcopy(ep->eth_packet.eth_header.src,
                                (unsigned char *)eth_rcv_pk.src.bytes, 6);

            hight_type = eth_rcv_pk.type[0];
            low_type = eth_rcv_pk.type[1];

            type = (hight_type << 8);
            type = type | low_type;

            ep->eth_packet.eth_header.type = type;

            pip = (struct ip *)ep->eth_packet.data;

                /* accept the frames for yourself or the broadcast frames */

            if( ( blkequ(ep->eth_packet.eth_header.dst, localaddr.bytes, 6)
                                                        == TCPIP_TRUE ) |
                ( blkequ(ep->eth_packet.eth_header.dst, eth_addr_brc, 6)
                                                        == TCPIP_TRUE ) ){
                /* call the function tcpip_rcvpack */

              tcpip_rcvpack(eth_if_no, (void *)ep, (unsigned)(EP_HLEN + pip->ip_len));
            }
        }
```

2.4.2 Interfacing an Ethernet and two ALSTOM WorldFIP pseudo drivers

This example shows how to find the interface type in the case of simultaneous use of Ethernet and ALSTOM WorldFIP interfaces.

Let us suppose that there are 1 Ethernet interface and 2 ALSTOM WorldFIP interfaces.

You have to create the **`eth_rcv_pack()`** function and call it periodically.

```
#include "tcpip.h"

……..

        /* global variables declared in the file that initialises the TCPIP stack module
          (see the example of how to initialise the TCP/IP Stack) */

extern    IF_DEV_TYPE    eth_dev_fd;
extern    IF_DEV_TYPE    fip1_dev_fd;
extern    IF_DEV_TYPE    fip2_dev_fd;

unsigned short    eth_if_no;

……..


/*-----------------------------------------------
 *  DEV_CONTROL : input/output control operations on a device
 *-----------------------------------------------
 */

DEV_CONTROL(IF_DEV_TYPE dev, int ctrl, void *args)
{
ETH_MSG_INIT    *eth_msg_init;

if( dev == eth_dev_fd){
                                        /* ETHERNET driver */
    eth_msg_init = (ETH_MSG_INIT *)args;

    switch(ctrl){

      case INIT_DEV :
          eth_if_no = eth_msg_init->if_number;
          tcpip_rcvpack = eth_msg_init->indication;
          break;

      case STOP_DEV :
          break;

      case FREE_MEM_DEV :
          tbx_FreeBufferInPool((TBX_USER_BUFFER *)args);
          break;

      default :
          break;
    }
}
else{
#if (WORLDFIP_PSEUDO_DRIVER == TCPIP_YES )
```

```
                                        /* ALSTOM WorldFIP pseudo driver */
  int         rv;

  rv = fip_ctrl(dev, ctrl, args);

  return(rv);

#endif}
}


/*-------------------------------------------------
 *  DEV_WRITE : write operations on a device
*-------------------------------------------------
 */

DEV_WRITE(IF_DEV_TYPE dev, void *xmit_buff, unsigned nbytes)
{
int                             rv;

if( dev = = eth_dev_fd){
                                  /* ETHERNET driver */

/*…….. */

              /* write the xmit_buff to your specific driver by a “write” function

                 rv = “ethernet_write(…..)”;  */

                   /*…….. */

                /* free the xmit_buff buffer */

                    tbx_FreeBufferInPool((TBX_USER_BUFFER *)xmit_buff);

                     return(rv);
}
else{
#if (WORLDFIP_PSEUDO_DRIVER = = TCPIP_YES )

                              /* ALSTOM WorldFIP pseudo driver */

  rv = fip_write(dev, xmit_buff, nbytes);

  return(rv);

#endif
}
}

/*****************************************************************************
 *eth_rcv_pack : read your specific ETHERNET device driver
*                      this function must be called periodically by the user
 *****************************************************************************
 */

void eth_rcv_pack()
{
  ETH_PACK          *ep;
  struct ip  *pip;
```

```
……..

  ep = (ETH_PACK *)tbx_AllocateBufferInPool(Net.eth_pool);
  if(ep == 0){
    fprintf(fd,"\n ERROR : Pool eth_pool \n");
    return;
  }
    pip = (struct ip *)ep->eth_packet.data;

                /* call your ETHERNET device driver read() function */
……..

              /* IF the received data is for yourself or a broadcast packet THEN
                    copy the received data in the memory pointed by ep
                 ELSE
                    ignore the received frame and free ep by the function
                    tbx_FreeBufferInPool((TBX_USER_BUFFER *)ep);
                 ENDIF  */

            /* call the function tcpip_rcvpack */

    tcpip_rcvpack(eth_if_no, (void *)ep, (unsigned)(EP_HLEN + pip->ip_len));
  }
}
```

Chapter 7 Software startup and stoppage

1. TCIP_START(): STARTUP PROCEDURE OF THE TCP/IP STACK MODULE

1.1 Description

The initialisation and start-up of the TCP/IP Stack module will be carried out by calling the **`tcpip_start()`** function.

The user must complete some structures that contain the parameters of each network interface.

Syntax

```
#include "tcpip.h"

tcpip_start(TBX_MEMORY_RN *tcpip_memory,
  IF_FIP *if_fip,
  IF_ETH *if_eth,
  SIGNALS_ADDR *signals_addr,
  ICMP_RESP_ADDR *icmp_resp_addr)
```

1.2 Inputs

`tcpip_memory`: pointer to a structure of the **`TBX_MEMORY_RN`** type (defined in the *Tools_Box/Tools/tbx.h* file) that allows you to define a memory region (if the operating system permits this) where will be allocated all the resources for this interface. If the operating system doesn't allow you to define a memory region then this parameter should be set to 0.

`if_fip`: pointer to a structure that describes the parameters of the WorldFIP interfaces; the **`IF_FIP`** type is described below. If you don't have WorldFIP interfaces then this parameter should be set to 0.

`if_eth`: pointer to a structure that describes the parameters of the Ethernet interfaces; the **`IF_ETH`** type is described below. If you don't have WorldFIP interfaces then this parameter should be set to 0.

`signals_addr`: pointer to a structure of type **`SIGNALS_ADDR`** (described below) which makes it possible to provide the **TCP/IP Stack** module with the callback addresses of the functions used for the Sequencer (see Chapter 5): **`user_signal_ip_rcv()`**,

`user_signal_tcp_in_rcv()` and **`user_signal_tcp_out_rcv().`**

`icmp_resp_addr`: pointer to a structure of the **`ICMP_RESP_ADDR`** type (described below) which makes it possible to inform the TCP/IP Stack module of the callback addresses of the functions:

- **`user_echo_reply()`**: user callback function that will be called by the TCP/IP Stack module whenever it receives an ECHO response (as a consequence of an ECHO request).

- **`user_mask_reply()`**: user callback function that will be called by the TCP/IP Stack module whenever it receives a Subnet Mask response (as a consequence of a Subnet Mask request).

Description of the `IF_FIP` and `IF_ETH` structures

```
typedef struct {
        unsigned short          nb_of_fip_ifs;
        FIP_IF_PARAM            **ptr_tab_fip_if_param;
}IF_FIP;


typedef struct {
        unsigned short          nb_of_eth_ifs;
        ETH_IF_PARAM            **ptr_tab_eth_if_param;
}IF_ETH;
```

`nb_of_fip_ifs:` number of WorldFIP interfaces.

`ptr_tab_fip_if_param`: pointer to an array of pointers to structures of the **`FIP_IF_PARAM`** type (see below).

`nb_of_eth_ifs`: number of Ethernet interfaces.

`ptr_tab_eth_if_param`: pointer to an array of pointers to structures of the **`ETH_IF_PARAM`** type (see below).

Description of the `FIP_IF_PARAM` and `ETH_IF_PARAM` structures

```
typedef struct {
        GENERAL                 *general;
        FIP_SPECIFIC            *fip_specific;
        void                    *fdm_ref;
        unsigned long           if_report;
}FIP_IF_PARAM;


typedef struct {
        GENERAL                 *general;
        ETH_SPECIFIC            eth_specific;
        unsigned long           if_report;
}ETH_IF_PARAM;
```

Note

All the fields of the above structures must be set by the user except the field **`if_report`** which will be set by the **TCP/IP Stack** module.

`general`: pointer to a structure of the **`GENERAL`** type (described below) that describes the general parameters of a WorldFIP or Ethernet interface.

`fip_specific`: pointer to a structure of the **`FIP_SPECIFIC`** type (described below) that describes the specific parameters of a WorldFIP interface.

`fdm_ref`: pointer that should be cast to a structure of the **`FDM_REF`** type defined in FIP DEVICE MANAGER User Manual [ALS 50278].

`eth_specific`: pointer to a structure of the **`ETH_SPECIFIC`** type (described below) that describes the specific parameters of an Ethernet interface.

`if_report`: return value of the interface initialisation; this field is set by the **TCP/IP Stack** module.

Description of the GENERAL structure

```
typedef struct {
    char                *if_name;
    unsigned short      if_number;
    IPaddr              if_ip_addr;
    int                 if_brc_type;
    int                 if_mtu;
    IF_DEV_TYPE         if_dev;
    unsigned short      if_arp_policy;
    TBX_MEMORY_RN       *if_memory;
}GENERAL;
```

Note

All the fields of the above structures must be set by the user except the field **`if_number`** which will be set by the **TCP/IP Stack** module.

If you don't know the IP address of an interface, you can get it if you call the **`tcpip_getaddr()`** function after the **`tcpip_start()`**function. The RARP mechanism will be started.

`if_name`: pointer to a string that contains the domain name of the interface.

`if_number`: interface number; this field is set by the TCP/IP Stack module.

`if_ip_addr`: point-to-point IP address.

`if_brc_type:` type of the broadcast address:

= 0 - all the bits that represent the host part of the IP address are set to 0

= 1 - all the bits that represent the host part of the IP address are set to 1

`if_mtu`: data link layer messages maximum size (*Maximum Transmit Unit*).

`if_dev`: **`dev`**: number of the device driver; the user has to set the **`IF_DEV_TYPE`** type in the *TCPIP_Stack/Device_Driver_Adapter/tcpipdrv.h* file if necessary.

`if_arp_policy`: type of the ARP policy concerning the contents of the ARP table:

= 0 - the content of this table is permanent

= 1 - the table will be Reset periodically (period = **`ARP_TIMEOUT`** to be set in the file *TCPIP_Stack/Configuration/tcpipcfg.h*) and ARP requests will be issued.

`if_memory`: pointer to a structure of the **`TBX_MEMORY_RN`** type (defined in the *Tools_Box/Tools/tbx.h*) that allows you to define a memory region (if the operating system permits this) where all the resources will be allocated for this interface; if the operating system doesn't allow you to define a memory region then this parameter should be set to 0.

Description of the `IPaddr` type

```
typedef        unsigned char IPaddr[IP_ALEN];
```

`IP_ALEN`: IP address length is the **`IP_ALEN`** parameter in the file *TCPIP_Stack/Configuration/tcpipcfg.h.*

Description of the `FIP_SPECIFIC` structure

```
typedef struct {
        FIP_ADDR            if_station_hard_addr;
        FIP_ADDR            if_brc_hard_addr;
        unsigned short      if_nb_of_rec_buf1;
        unsigned short      if_nb_of_rec_buf2;
        unsigned short      if_msg_type;
        unsigned short      if_channel_nr;
        unsigned short      if_ident;
        void                (*addr_user_signal_fip_fatal_error)(unsigned long, int);
        void                (*addr_user_signal_fip_warning)(unsigned long, int);
}FIP_SPECIFIC;
```

`if_station_hard_addr`: point-to-point (or sometimes called "local") hardware address of the interface (3 bytes)

`if_brc_hard_addr`: broadcast hardware address of the interface (3 bytes)

`if_nb_of_rec_buf1`: number of reception buffers for the point-to-point address (to be allocated by the WorldFIP device driver)

`if_nb_of_rec_buf2`: number of reception buffers for the broadcast address (to be allocated by the WorldFIP device driver)

`if_msg_type`: WorldFIP messaging type:

= 0 - aperiodical messaging,

= 1 - periodical messaging

`if_channel_nr`: channel number:

= 0 in the case of aperiodical messaging,

= 1 to 8 in the case of periodical messaging.

`if_ident`: identifier of the produced variable in the case of periodical messaging; should be set to 0 in the case of aperiodical messaging.

`addr_user_signal_fip_fatal_error`: address of a callback function that will be called whenever the WorldFIP device driver detects a communication error.

`addr_user_signal_fip_warning`: address of a callback function that will be called whenever the WorldFIP device driver detects a communication warning.

Description of the `ETH_SPECIFIC` structure

```
typedef struct {
        ETH_ADDR            if_station_hard_addr;
        ETH_ADDR            if_brc_hard_addr;
        unsigned short      if_nb_of_rec_buf;
        unsigned short      if_size_of_rec_buf;
}ETH_SPECIFIC;
```

`if_station_hard_addr`: point-to-point (or sometimes called local) hardware address of the interface (6 bytes)

`if_brc_hard_addr`: broadcast hardware address of the interface (6 bytes)

`if_nb_of_rec_buf`: number of reception buffers

`if_size_of_rec_buf`: size of a reception buffer.

Description of the `SIGNALS_ADDR` structure

```
typedef struct {
        void    (*addr_user_signal_tcp_in_rcv)(void);
        void    (*addr_user_signal_tcp_out_rcv)(void);
        void    (*addr_user_signal_ip_rcv)(void);
}SIGNALS_ADDR;
```

`addr_user_signal_tcp_in_rcv`: address of a user callback function; this function is called by the **TCP/IP Stack** module whenever the IP layer writes a message in the shared message queue between the IP layer and **`TCP_IN`** layer.

`addr_user_signal_tcp_out_rcv`: address of a user callback function; this function is called by the **TCP/IP Stack** module whenever the **SockAPI** module writes a message in the shared message queue between the SockAPI module and **`TCP_OUT`** layer.

`addr_user_signal_ip_rcv`: address of a user callback function; this function is called by the **TCP/IP Stack** module whenever the TCP, UDP, ICMP or Network Interface layers write a message in the shared message queue between these layers and the IP layer.

Note

The user has to implement these callback functions; he has to empty the shared queues (for details see Chapter 2, Section 4 TCP/IP Sequencer).

Description of the `ICMP_RESP_ADDR` structure

```
typedef struct {
        void    (*addr_user_echo_reply)(struct ip *);
        void    (*addr_user_mask_reply)(struct ip *);
}ICMP_RESP_ADDR;
```

`addr_user_echo_reply`: address of a user callback function; this function is called by the **TCP/IP Stack** module after receiving an ECHO response as a consequence of a call by the user of the function **`tcpip_echo_rq().`** If this address is null it will be communicated to the TCP/IP Stack module as a parameter when the user calls the function **`tcpip_echo_rq()`**.

`addr_user_mask_reply`: address of a user callback function; this function is called by the **TCP/IP Stack** module after receiving a SUBNET MASK response as a consequence of a call by the user of the function **`tcpip_get_subnet_mask()`**.

1.3 Outputs

Value	Description
0	OK
1	if the **nb_of_fip_ifs** field of the **if_fip** parameter is greater than **NMAXFIP** (defined in the *TCPIP_Stack/Configuration/tcpipcfg.h* file).
2	if the **nb_of_eth_ifs** field of the **if_eth** parameter is greater than **NMAXETH** (defined in the *TCPIP_Stack/Configuration/tcpipcfg.h* file).
3	if the **nb_of_fip_ifs** field of the **if_fip** parameter is 0 and the **nb_of_eth_ifs** field of the **if_eth** parameter is 0.
4	if there is a memory problem.
5	if there is an initialisation problem for one of the timers used by the **TCP/IP Stack** module
6	if there is an initialisation problem for at least one WorldFIP device driver; a detailed description of the error is provided in the **if_report** field of the **FIP_IF_PARAM** structure; the user should check all of these structures to find the faulty device.
7	if there is an initialisation problem for at least one Ethernet device driver; a detailed description of the error is provided in the **if_report** field of the **ETH_IF_PARAM** structure; the user should check all of these structures to find the faulty device.
8	if there is an initialisation problem of the critical region in the **Tools Box** module.

1.4 Example

Let us suppose that there are 1 Ethernet interface and 2 ALSTOM WorldFIP interfaces.

```
#define ETH_ADDR_BRC            {0xff, 0xff, 0xff, 0xff, 0xff, 0xff}    /*hardware broadcast address*/
#define ETH_ADDR_ST1            {0x8, 0, 0x20, 0x1f, 0x44,0x67} /*hardware  point to point
                                                                  address */
#define IP_ST1_ETH                      "192.9.230.1"                           /* IP address */


static ETH_SPECIFIC     eth_specific =      {
                                                ETH_ADDR_ST1,
                                                ETH_ADDR_BRC,
                                                NB_OF_ETH_BUF,
                                                SIZE_OF_ETH_BUF
                                            };


static GENERAL    eth_general = {
                                            "eth1",                     /* if name */
                                            0,                          /*if no - is set by the TCPIP Stack module*/
                                            {0, 0, 0, 0},               /* IP addr - set by calling sockapi_dot2ip()*/
                                            0,                          /* broadcast type : specify the "host"
                                                                    value of an IP address :
                                                                                        0 = aperiodical,
                                                                                          1 = periodical */
                                            MAX_ETH_MTU,   / Maximum Transmit Unit */
                                            0,                          /* device number */
                                            0,                  /* arp policy :
                                                                        0 → permanent entries in ARP table,
                                                                        1 →  the ARP table is updated periodically*/
                                            0                   /* TBX_MEMORY_RN */
                                    };



static ETH_IF_PARAM     eth_if_param ={
                                                &eth_general,
                                                &eth_specific,
                                                0                               /* if. report */
                                            };
```

Figure 7.1 - Definition of the Ethernet interface

```
                                /* WorldFIP Interface 1 */

                                    /*LSAP, Station, Segment */
#define FIP_ADDR_BRC        {0xff, 0xfe, 0x0}          /*hardware broadcast address*/

                                  /*LSAP, Station, Segment */
#define HWAD_ST1_FIP1  {0x01, 0x01, 0x0}          /* hardware point to point address*/

#define IP_ST1_FIP1                  "192.9.250.1"              /* IP address */

            /* user callback functions to treat the FIP DEVICE MANAGER errors and warnings */

static void  wfip1_drv_errors(unsigned long, int);
static void  wfip1_drv_warnings(unsigned long, int);

static FIP_SPECIFIC          fip_specific1 ={
                                        HWAD_ST1_FIP1,       /* hardware point to point address */
                                        FIP_ADDR_BRC,        /* hardware broadcast address */
                                        NB_OF_FIP_BUF/4,     /* number of buffers to be allocated for
                                                             the point to point address pool by the
                                                             WorldFIP pseudo-driver */
                                        NB_OF_FIP_BUF/4,     /* number of buffers to be allocated for
                                                             the broadcast address pool by the
                                                             WorldFIP pseudo-driver */
                                        0,                   /* type of message :
                                                                    0 = aperiodical,
                                                                    1 = periodical */
                                        0,                   /* channel number(1 to 8) for a
                                                             periodical channel */
                                        0,                   /* Identifier if periodical channel */
                                        wfip1_drv_errors,  /* User function to treat FDM errors */
                                        wfip1_drv_warnings      /* User function to treat FDM warnings */
                                    };

static GENERAL    fip_general1 = {
                                        "wfip1",  /* if name */
                                        0,                  /*if no - is set by the TCPIP Stack module*/
                                        {0, 0, 0, 0},       /* IP addr - set by calling sockapi_dot2ip()*/
                                        0,                  /* broadcast type : specify the "host"
                                                                value of an IP address :
                                                                        0 = aperiodical,
                                                                          1 = periodical */
                                        MAX_FIP_MTU, /* Maximum Transmit Unit */
                                        0,                  /* device number */
                                        0,                  /* arp policy :
                                                                    0 → permanent entries in ARP table,
                                                                    1 →  the ARP table is updated
                periodically*/
                                                0                   /* TBX_MEMORY_RN */
                                        };

static FIP_IF_PARAM          fip_if_param1 = {
                                                &fip_general1,
                                                &fip_specific1,
                                                0,                          /* FDM interface ref */
                                                0                           /* interface initialisation
report */                                    };
```

Figure 7.2 - Definition of the first WorldFIP interface

```
                                /* WorldFIP Interface 2 */

                                  /*LSAP, Station, Segment */
#define HWAD_ST1_FIP2  {0x01, 0x1F, 0x1}          /* hardware point to point address*/

#define IP_ST1_FIP2                "192.9.254.31"          /* IP address */

            /* user callback functions to treat the FIP DEVICE MANAGER errors and warnings */

static void  wfip2_drv_errors(unsigned long, int);
static void  wfip2_drv_warnings(unsigned long, int);


static FIP_SPECIFIC        fip_specific2 ={
                                    HWAD_ST1_FIP2,      /* hardware point to point address */
                                    FIP_ADDR_BRC,       /* hardware broadcast address */
                                    NB_OF_FIP_BUF/4,    /* number of buffers to be allocated for
                                                        the point to point address pool by the
                                                        WorldFIP pseudo-driver */
                                    NB_OF_FIP_BUF/4,    /* number of buffers to be allocated for
                                                        the broadcast address pool by the
                                                        WorldFIP pseudo-driver */
                                    0,                  /* type of message :
                                                                0 = aperiodical,
                                                                1 = periodical */
                                    0,                  /* channel number(1 to 8) for a
                                                        periodical channel */
                                    0,                  /* Identifier if periodical channel */
                                    wfip2_drv_errors,   /* User function to treat FDM errors */
                                    wfip2_drv_warnings  /* User function to treat FDM warnings */
                                };

static GENERAL    fip_general2 = {
                                    "wfip2",   /* if name */
                                    0,              /*if no - is set by the TCPIP Stack module*/
                                    {0, 0, 0, 0},   /* IP addr - set by calling sockapi_dot2ip()*/
                                    0,              /* broadcast type : specify the "host"
                                                        value of an IP address :
                                                                0 = aperiodical,
                                                                1 = periodical */
                                    MAX_FIP_MTU, /* Maximum Transmit Unit */
                                    0,              /* device number */
                                    0,              /* arp policy :
                                                        0 → permanent entries in ARP table,
                                                        1 →  the ARP table is updated
            periodically*/
                                    0               /* TBX_MEMORY_RN */
                                    };

static FIP_IF_PARAM        fip_if_param2 = {
                                        &fip_general2,
                                        &fip_specific2,
                                        0,                      /* FDM interface ref */
                                        0                       /* interface initialisation report */
                                    };
```

Figure 7.3 - Definition of the second WorldFIP interface

Definition of the `IF_ETH` and `IF_FIP` parameters

```
static ETH_IF_PARAM     *tab_eth_if_param[1] = {
                                                &eth_if_param
                                            };
static IF_ETH           if_eth = {
                                    1,                      /*nb of ETHERNET ifs */
                                    tab_eth_if_param
                                };


static FIP_IF_PARAM     *tab_fip_if_param[2] = {
                                                &fip_if_param1,
                                                &fip_if_param2
                                            };
static IF_FIP           if_fip = {
                                    2,  /* nb. of FIP interfaces */
                                    tab_fip_if_param
                                };
```

Definition of the user callback functions used for the TCP/IP Sequencer

```
static void             tcp_in_rcv();
static void             tcp_out_rcv();
static void             ip_rcv();

static SIGNALS_ADDR     signals_addr =   {
                                                tcp_in_rcv,
                                                tcp_out_rcv,
                                                ip_rcv
                                            };
```

Definition of the user callback functions used to treat the responses of an ECHO request (call of `tcpip_echo_rq()`) and of a "Get Subnet Mask" request (call of `tcpip_get_subnet_mask()`).

```
static void             user_echo_reply(struct ip *);
static void             user_mask_reply(struct ip *);

static ICMP_RESP_ADDR   icmp_resp_addr = {
                                                user_echo_reply,
                                                user_mask_reply
                                            };
```

Call of the `tcpip_start()` function

```
#include “sockapi.h”

……

                                /* device drivers handles */
IF_DEV_TYPE     eth_dev_fd;
IF_DEV_TYPE     fip1_dev_fd;
IF_DEV_TYPE     fip2_dev_fd;
                                /* interface numbers */
unsigned short        eth_if_no;
unsigned short        fip1_if_no;
unsigned short        fip2_if_no;
                                /* FIP DEVICE MANAGER context references for each WorldFIP interface */
static FDM_REF  *FULLFIP1;                  /* see FDM User Manual */
static FDM_REF  *FULLFIP2;

……

            main()
            {
                int       rv;

                ....................................................................................

                        /*  open yourETHERNET  device driver by a “open”  function and get the
                          device driver handle eth_dev_fd*/
                ....................................................................................

                        /* set the fields if_dev et if_ip_addr of eth_general structure*/

                eth_general.if_dev = (IF_DEV_TYPE)eth_dev_fd;
                sockapi_dot2ip( eth_general.if_ip_addr,  IP_ST1_ETH) ;
                ....................................................................................



                         /* set the WorldFIP pseudo-drivers handles */
                fip1_dev_fd = 1;
               fip2_dev_fd = 2;

                        /* set the fields if_dev et if_ip_addr of fip_general1 and fip_general1
            structures*/

                fip_general1.if_dev = (IF_DEV_TYPE)fip1_dev_fd;
                sockapi_dot2ip (fip_general1.if_ip_addr,  IP_ST1_FIP1 );

                fip_general2.if_dev = (IF_DEV_TYPE)fip2_dev_fd;
                sockapi_dot2ip( fip_general2.if_ip_addr,  IP_ST1_FIP2 );

                        /* initialise the FULLFIP2 components of the  2 WorldFIP interfaces using
                          fdm_initialize_network() function of FDM */

                FULLFIP1 = fdm_initialize_network(.......);
```

```
                FULLFIP2 = fdm_initialize_network(.......);

                fip_if_param1.fdm_ref = FULLFIP1;
                fip_if_param2.fdm_ref = FULLFIP2;

            ..........................................................................................

                rv = tcpip_start(0, &if_fip, &if_eth, &signals_addr, &icmp_resp_addr);

                if(rv != 0){
                    printf("\n tcpip_start : Error = %d\n", rv);
                    tcpip_stop();
                }
                else{
                        /* get and memorise the interface numbers; these numbers
                           are set by the TCPIP Stack module and will be  used by some
                           TCPIP Stack module functions */

                    eth_if_no = eth_general.if_number;
                    fip1_if_no = fip_general1.if_number;
                    fip2_if_no = fip_general2.if_number;

                ..........................................................................

            }
}
```

2. SOCKAPI_START(): STARTUP PROCEDURE OF THE SOCKAPI MODULE

2.1 Description

The **`sockapi_start`** function allows a user to start the **SockAPI** library services. Before starting the **SockAPI** library services you should start the **TCP/IP - WorldFIP** library (see [ALS 50278]).

Function syntax:

```
#include "sockapi.h"
int   sockapi_start( TBX_MEMORY_RN *sockapi_memory );
```

2.2 Inputs

`sockapi_memory`: pointer to a structure of the **`TBX_MEMORY_RN`** type (defined in the *Tools_Box/Tools/tbx.h* file) that allows you to define a memory region (if the operating system permits this) where will be allocated all the resources for this module; if the operating system doesn't allow you to define a memory region then this parameter should be set to 0.

2.3 Outputs

Value	Definition
0	OK
-1	error; in this case call the **`sockapi_get_last_error`** function to get the last error

2.4 Fault messages

The return value of the **`sockapi_get_last_error`** function is **`ESOCKAPI_ALLREADYINITIALISED`**: it attempts to call the **`sockapi_start`** function more than once.

2.5 Performance

This function is a non blocking function executed in the context of the user thread or process.

3. TCPIP_STOP(): STOPPAGE PROCEDURE OF THE TCP/IP STACK MODULE

When the user doesn't need to use the **TCP/IP Stack** module anymore, he can call the **`tcpip_stop()`** function. All the resources allocated by the **TCP/IP Stack** module will be freed.

Before proceeding with the stop request, the user must make sure that all his requests in progress have been finished. Then he must close all the open connections (UDP and TCP) and block the TCP/IP Sequencer tasks.

Note

In case of use of TCP protocol, after the closing of the **stream sockets**, it is necessary to wait a time greater than **`TCP_TWOMSL`** (defined in the *TCPIP_Stack/Configuration/tcpipcfg.h* file), before proceeding to the stop request.

Function syntax

```
#include "tcpip.h"
void tcpip_stop()
```

4. SOCKAPI_STOP(): STOPPAGE PROCEDURE OF THE SOCKAPI MODULE

4.1 Description

The **`sockapi_stop()`** function allows a user to stop the **SockAPI** library services.

Function syntax:

```
#include "sockapi.h"
int   sockapi_stop();
```

4.2 Outputs

Value	Description
0	OK
-1	error; in this case call the **`sockapi_get_last_error`** function to get the last error

4.3 Fault messages

The returning value of the **`sockapi_get_last_error`** function is **`ESOCKAPI_NOTINITIALISED`**: It attempts to call a library function without first calling previously the **`sockapi_start`** function.

4.4 Performance

This function is a non blocking function executed in the context of the user thread or process.

Chapter 8 Socket functions

Generally a network application consists of a client and a server component. Frequently these components are isolated in separate programs; sometimes they are integrated in a single application. Generally they are implemented on different stations.

The following state diagrams illustrate the state transactions for setting up Clients and Servers that use **stream sockets**:

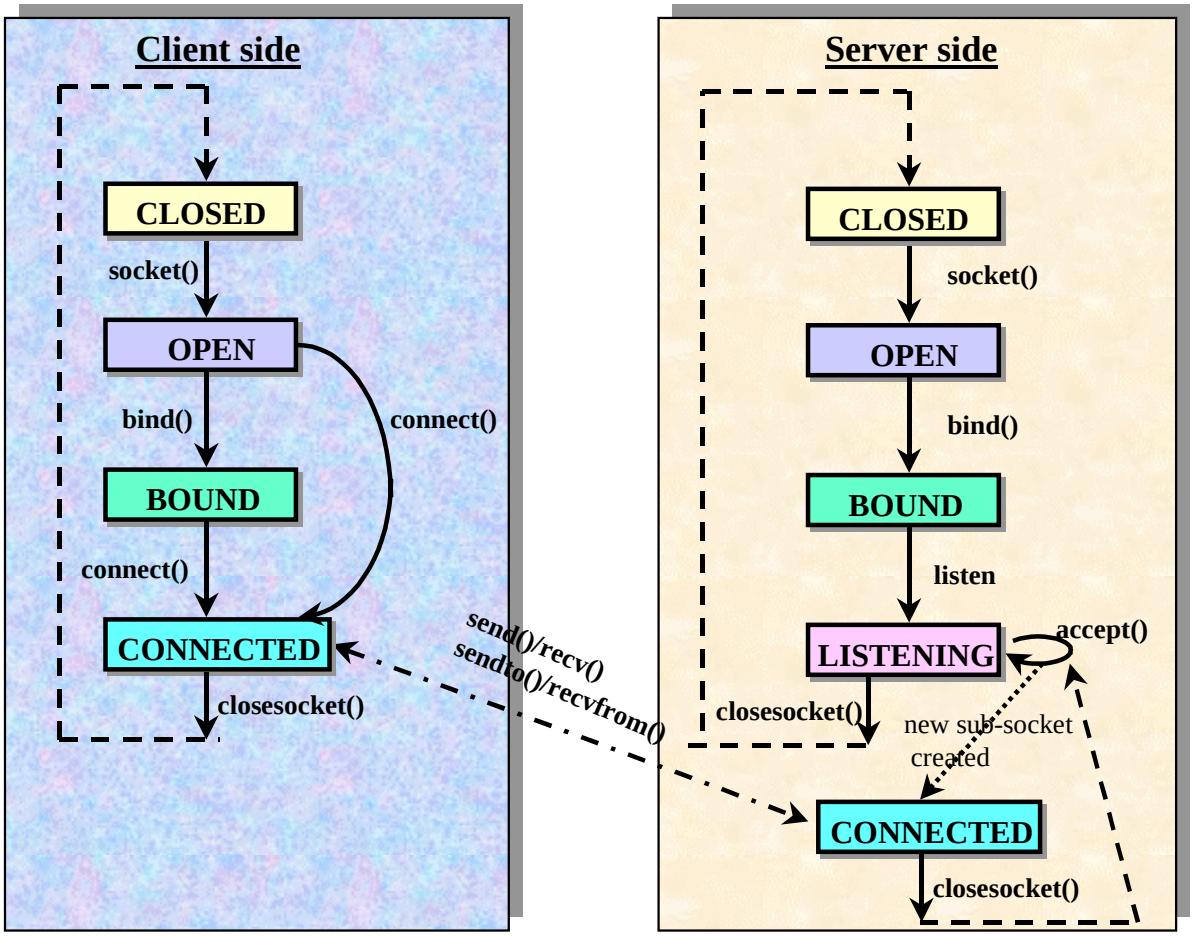


Figure 8.1 - State transactions for setting up clients and servers that use stream sockets

The **socket** call creates an endpoint for both client and server and returns an integer which uniquely identifies the created socket.

The **`bind`** command is used to bind the socket to a locally specified endpoint address.

The local endpoint address is comprised of two elements: the IP address and the port number for the socket.

The user has the possibility to substitute the value **`INADDR_ANY`** in place of a valid local IP address. In this case the TCP/IP Suite will find itself the appropriate interface. If you provide a value of 0 for the **`sin_port`** field of the **`sockaddr_in`** structure then the TCP/IP Suite will allocate a port number for your application.

Client-side connection set up

Once the socket is bound, the **`connect`** command will establish a connection with a remote endpoint. Although the client may choose to call **`bind`**, it is not necessary to do so. Calling **`connect`** with an unbound socket will simply force the TCP/IP Suite to choose an IP interface and a unique port number and will mark the socket as bound.

Server-side connection set up

Once the socket is bound, the **`listen`** command will set up a queue of incoming connection requests. The **`accept`** command removes the next connection request from the queue created by the **`listen`** function, creates a sub-socket for the request and returns the identifier of the sub-socket. The Server will interact with the Client on the new created socket.

Sending and receiving data

Once the application has successfully established a socket connection, it is ready to start transferring data over the connection. Normally an application sends data using the **`send`** function and receives data using the **`recv`** function, but the application can also use the functions **`sendto`** and **`recvfrom`**.

Terminating a connection

To terminate the connection the application has to call **`closesocket()`**.

The following state diagrams illustrate the state transactions for setting up clients and servers that use **datagram sockets**:

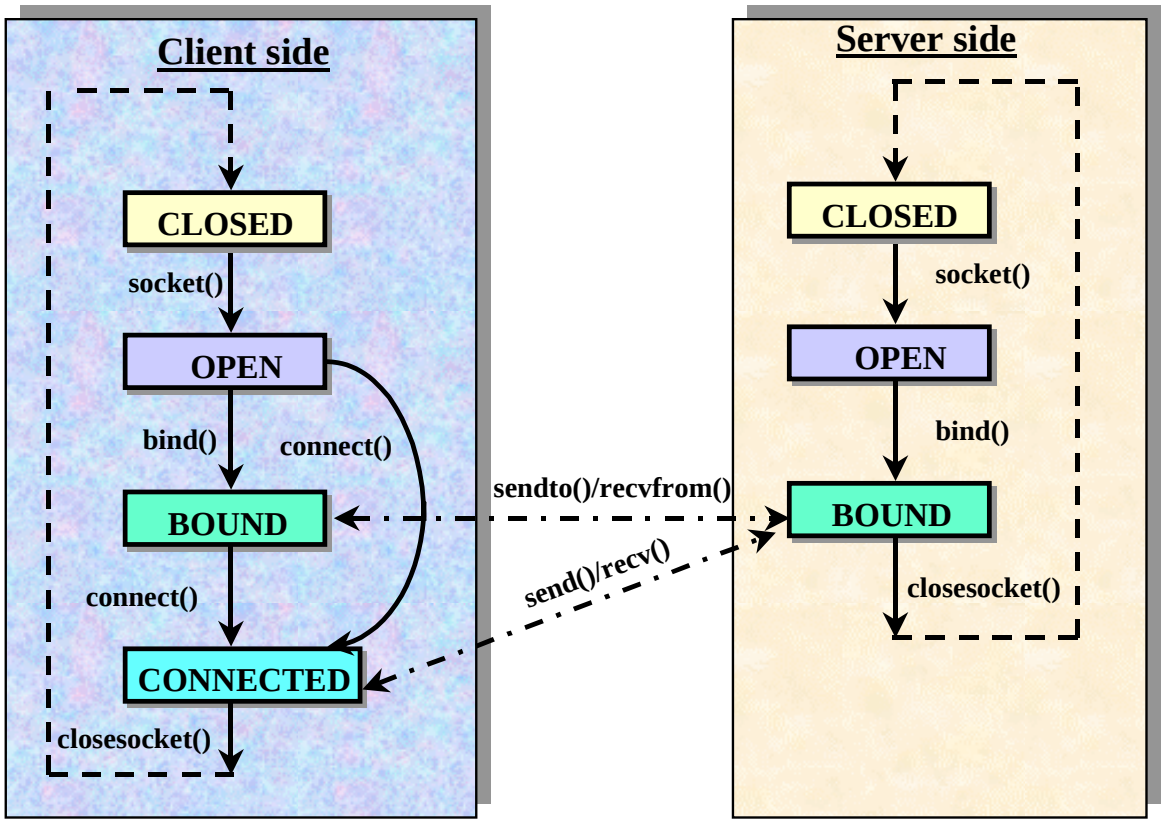


Figure 8.2 - Transactions for setting up clients and servers that use datagram sockets

Since the communication in **datagram sockets** is connectionless, it is not necessary to use the functions used for the creation of a connection, namely **`connect()`**, **`listen()`** and **`accept()`**.

A client may choose to **`connect()`** the datagram socket. Connecting a datagram socket will cause all sends (using **`send()`**) to go to the connected address, and any datagrams received (using **`recv()`**) from a remote address different than the connected address are discarded by the system.

Transmitting data on a datagram socket is not reliable. This means that if an application attempts to send data to another application, the system does nothing to guarantee that the data will actually be delivered to the remote application.

Datagram sockets are buffer-oriented; that means that the system does not coalesce data when sending it. For example if an application makes 10 calls to **`sendto()`** or **`send()`** with a buffer of 300 bytes, the remote application will need to perform 10 calls to **`recvfrom()`** or **`recv()`** with a buffer of at least 300 bytes to receive all the data.

1. ANOMALY REPORTS

When any of the standard functions fails, the application can call the function **`sockapi_get_last_error`** to get the last error.

The errors are defined in the file *SOCKAPI/sockapi.h*:

Error code	Description
ESOCKAPI_NOTINITIALISED	attempts to call a library function without first calling the **`sockapi_start`** function
ESOCKAPI_ALLREADYINITIALISED	attempts to call more than one **`sockapi_start`** function
ESOCKAPI_FAMILYNOSUPPORT	if the family is not **`AF_INET`** or **`PF_INET`** used for the Internet protocol
ESOCKAPI_TYPENOSUPPORT	if the socket type is not **`SOCK_STREAM`** (for a stream socket) or **`SOCK_DGRAM`** (for a datagram socket)
ESOCKAPI_PROTOCOLNOSUPPORT	if the protocol is not supported
ESOCKAPI_NOBUFS	no more buffers are available
ESOCKAPI_NOTSOCK	incorrect socket number
ESOCKAPI_ISCONN	the socket is already connected
ESOCKAPI_INVAL	incorrect argument or the socket is already bound
ESOCKAPI_FAULT	incorrect address length
ESOCKAPI_ADDRINUSE	the local port is already used by another process
ESOCKAPI_CONNREFUSED	connection refused by the remote machine
ESOCKAPI_TIMEDOUT	attempt to connect; time-out without establishing the connection
ESOCKAPI_OPNOTSUPP	the socket type does not support a ‘listen’ operation
ESOCKAPI_NOTBOUND	the socket is not bound
ESOCKAPI_NOCONREQ	the socket is non blocking and no connection requests are present to be accepted
ESOCKAPI_NOLISTEN	*listen* operation was not invoked prior to *accept* operation
ESOCKAPI_INTERNALTCP	internal error in the TCP layer
ESOCKAPI_NOTCONN	the socket is not connected
ESOCKAPI_OPFORBIDDEN	the operation is forbidden
ESOCKAPI_MSGSIZE	the message is too large for the socket
ESOCKAPI_CONNRESET	the connection is closed by the remote side
ESOCKAPI_NOMEMORY	memory allocation error
ESOCKAPI_MAXRETRANS	give up sending the message; the message was repeated for the **`TCP_MAXRETRIES`** parameter of the TCP layer (defined in *TCPIP_Stack/Configuration/tcpipcfg.h* file)
ESOCKAPI_NODATA	the socket has no data and the socket was created as non blocking
ESOCKAPI_SOCKINUSE	this socket is in use as a non-connected socket
ESOCKAPI_EINTR	the blocking call was cancelled via a close of the socket

2. BASIC STRUCTURES

```
struct    sockaddr {
          u_short       sa_family;
          char          sa_data[14];
};

struct    in_addr {
          union {
                struct { u_char s_b1, s_b2, s_b3, s_b4;} S_un_b;
                struct { u_short s_w1, s_w2;} S_un_w;
                u_long  S_addr;
          }S_un;
#define   s_addr        S_un.S_addr         /*should be used for all code*/
#define   s_host        S_un.S_un_b.s_b2    /* OBSOLETE: host on imp */
#define   s_net         S_un.S_un_b.s_b1    /* OBSOLETE: network */
#define   s_imp         S_un.S_un_w.s_w2    /* OBSOLETE: imp */
#define   s_impno       S_un.S_un_b.s_b4    /* OBSOLETE: imp # */
#define   s_lh          S_un.S_un_b.s_b3    /* OBSOLETE: logical host */
};

struct    sockaddr_in {
          short                 sin_family;
          u_short               sin_port;
          struct in_addr        sin_addr;
          char                  sin_zero[8];
};

#define   IP_ADDLEN             4
typedef   unsigned char         IPAddr[IP_ADDLEN]
```

Figure 8.3 – Basic structures of the socket functions

The **sockaddr** structure (defined in *SOCKAPI/socket.h* file) is used to specify a local or a remote endpoint address to which a socket can be connected.

In the Internet address family the **sockaddr_in** structure (defined in *SOCKAPI/in.h* file) is used to store the endpoint address information. This structure normally is used to cast the **sockaddr** type for the functions which require it. The value for the **sin_family** field should be **AF_INET**. The **sin_port** field is the port of the endpoint, and the **sin_addr** field is the IP address of the endpoint. The last **sin_zero** field is used to pad the remaining 8 bytes of the endpoint address with the value 0.

IPAddr (defined in *SOCKAPI/sockapi.h* file) is a table of **IP_ADDLEN** characters that defines an IP address.

3. SOCKET()

3.1 Description

The **`socket`** function allows you to create a socket. A socket defines a bi-directional endpoint for communication between processes. Bi-directional implies that an application can transmit and receive data through these connections.

A socket has three primary components:

- the interface to which it is bound (specified by an IP address),
- the port number to which an application will send or receive data,
- the type of socket: stream or datagram.

The function returns an identifier for the socket, that will be used as a parameter each time that an operation on a socket will be executed.

By default the created socket is a **blocking socket**. The operations on this socket are blocking; i.e. a function is invoked which does not return until the associated operation is completed.

To set the socket as **non blocking** you have to use the **`sockapi_ioctl`** function after the **socket** function.

Function syntax:

```
#include "sockapi.h"
int   socket( int family, int type, int protocol );
```

3.2 Inputs

`family`: protocol family (defined in the so*cket.h* file). The protocol family is the same as the address family; the only currently supported family is **`PF_INET`** or **`AF_INET`** for the Internet protocols

`type`: socket type:

- SOCK_STREAM for a stream socket (connection-oriented socket).
- SOCK_DGRAM for a datagram socket (connectionless socket)

`protocol`: should be 0

3.3 Outputs

The identifier of the socket

-1 → error; in this case call the function **`sockapi_get_last_error`** to get the last error

3.4 Fault messages

The returning values of the function **`sockapi_get_last_error`** are:

Error code	Description
`ESOCKAPI_NOTINITIALISED`	attempts to call a library function without first calling a **`sockapi_start`** function
`ESOCKAPI_FAMILYNOSUPPORT`	if the family is not **`AF_INET`** or **`PF_INET`** used for the Internet protocol
`ESOCKAPI_TYPENOSUPPORT`	if the type of the socket is not **`SOCK_STREAM`** (for a stream socket) or **`SOCK_DGRAM`** (for a datagram socket)
`ESOCKAPI_PROTOCOLNOSUPPORT`	if the parameter **protocol** is not 0; the specified protocol is not supported
`ESOCKAPI_NOBUFS`	no more socket descriptors are available; the maximum number that can be created is given by **`MAX_SOCKETS`** defined in *sockapi.h*

3.5 Performance

This function is a non blocking function and is executed in the context of the user thread or process.

4. BIND()

4.1 Description

The **bind** function allows you to associate the freshly created socket identifier to a local endpoint address. The local endpoint address comprises the IP address and the port number.

Function syntax:

```
#include "sockapi.h"
int   bind( int socket, struct sockaddr *localaddr, int addrlen );
```

4.2 Inputs

socket: a socket identifier created by the **socket** function

localaddr: pointer to a **sockaddr** structure that specifies the local IP address and the local port number

addrlen: size of the address structure (should be size of (**sockaddr_in**))

4.3 Outputs

0 → OK

-1 → error; in this case call the **sockapi_get_last_error** function to get the last error

4.4 Fault messages

The returning values of the **sockapi_get_last_error** function are:

Error code	Description
ESOCKAPI_NOTINITIALISED	attempts to call a library function without first calling the **sockapi_start** function
ESOCKAPI_FAMILYNOSUPPORT	if the family is not **AF_INET** or **PF_INET** used for the Internet protocol
ESOCKAPI_NOTSOCK	incorrect socket identifier
ESOCKAPI_ISCONN	the socket is already connected
ESOCKAPI_INVAL	incorrect argument (**localaddr**) or the socket is already bound
ESOCKAPI_FAULT	incorrect address length
ESOCKAPI_ADDRINUSE	the local port is already used by another process

4.5 Performance

This function is a non blocking function executed in the context of the user thread or process.

5. SOCKAPI_IOCTL()

5.1 Description

The **`sockapi_ioctl`** function allows you to set:

- the operating mode for a stream socket or a datagram socket: blocking or non blocking,
- for the stream socket:
  - the Nagle algorithm for the TCP layer,
  - the buffered delivery from the TCP layer.

A stream socket or a datagram socket has 2 operating modes:

- **blocking**: the operations on blocking sockets are blocking, i.e. invoking a function which does not return until the associated operation is completed.
- **non blocking**: the operations on these sockets are non blocking. This socket will be called a non blocking socket.

For a **stream socket** the Nagle algorithm is used by the TCP layer to reduce the number of small packets sent by a host by delaying the transmission of the acknowledgement.

It delays the acknowledgement in order to possibly have data going in the same direction as the acknowledgement, so the acknowledgement can be sent along with the data. (the value of the delay is set to **`TCP_ACKDELAY`** (defined in *TCPIP_Stack/Configuration/tcpipcfg.h* file).

For a **stream socket** the TCP layer delivers the received bytes in two manners:

- **buffered delivery** permits TCP layer to deliver the receiving bytes only when sufficient data has been received (only when the buffer contains at least **length** bytes specified by the functions **`recv`** or **`recvfrom`**). However if the data has arrived with the PUSH bit set (see TCP header) then the data will be delivered immediately.
- **unbuffered delivery** permits TCP layer to deliver as much information as is currently available, up to the size of **length** bytes specified by the functions **`recv`** or **`recvfrom`**.

By default a created:

- stream socket or datagram socket is a blocking socket,
- stream socket has:
  - the Nagle algorithm disabled,
  - the buffered delivery disabled.

Note

The **`sockapi_ioctl`** function is accepted only by a main socket; all the sub-sockets resulting from a **`accept`** function will inherit the same properties as the main socket.

Function syntax:

```
#include "sockapi.h"
int   sockapi_ioctl( int socket, int command, char *arg );
```

5.2 Inputs

`socket`: a socket identifier created by the **`socket`** function

`command`: it accepts the following values:

- **`FIONBIO`** for setting the operating mode of the socket,
- **`TCPOPTS`** for setting the Nagle algorithm and/or the buffered delivery.

`arg`: a pointer to a command parameter. If the command is:

- **`FIONBIO`** then **`arg`** points to an integer with the following values:

  = 0 → sets the socket as blocking

  > 0 → sets the socket as non-blocking

- **`TCPOPTS`** then **`arg`** points at an integer with the following values:
  - **`TCBF_DELACK`** enables the Nagle algorithm,
  - **`TCBF_BUFFER`** enables the buffered delivery,
  - **`TCBF_DELACK | TCBF_BUFFER`** enables the Nagle algorithm and the buffered delivery.

5.3 Outputs

0 → OK

-1 → error; in this case call the **`sockapi_get_last_error`** function to get the last error.

5.4 Fault messages

The returning values of the **`sockapi_get_last_error`** function are:

Error code	Description
`ESOCKAPI_NOTINITIALISED`	attempts to call a library function without first calling the **`sockapi_start`** function
`ESOCKAPI_NOTSOCK`	incorrect socket identifier, or a sub-socket
`ESOCKAPI_ISCONN`	the socket is already connected
`ESOCKAPI_INVAL`	incorrect argument
`ESOCKAPI_OPNOTSUPP`	if trying to apply the **`TCPOPTS`** command on a **datagram socket**

5.5 Performance

This function is a non-blocking function and is executed in the context of the user thread or process.

6. CONNECT()

6.1 Description

The **`connect`** function allows the caller to specify the remote endpoint address for a previously created socket.

If the socket is a **stream socket** that uses TCP, a 3-way handshake protocol to establish the connection will be initiated. If the socket is a **datagram socket** that uses UDP, **`connect`** only specifies the remote endpoint, but does not transfer any datagrams to it.

Function syntax:

```
#include "sockapi.h"
int   connect( int socket, struct sockaddr *remoteaddr, int addrlen );
```

6.2 Inputs

`socket`: a socket identifier created by the **`socket`** function

`remoteaddr`: pointer to a sockaddr structure that specifies the remote IP address and the remote port number

`addrlen`: size of the address structure (should be size of (**`sockaddr_in`**))

6.3 Outputs

0 → OK

-1 → error; in this case call the **`sockapi_get_last_error`** function to get the last error.

6.4 Fault messages

The returning values of the **`sockapi_get_last_error`** function are:

Error code	Description
`ESOCKAPI_NOTINITIALISED`	attempts to call a library function without first calling the **`sockapi_start`** function
`ESOCKAPI_FAMILYNOSUPPORT`	if the family is not **`AF_INET`** or **`PF_INET`** used for the Internet protocol
`ESOCKAPI_NOTSOCK`	incorrect socket identifier
`ESOCKAPI_ISCONN`	the socket is already connected
`ESOCKAPI_INVAL`	incorrect argument (**`remoteaddr`**)
`ESOCKAPI_FAULT`	incorrect address length
`ESOCKAPI_CONNREFUSED`	connection refused by the remote machine
`ESOCKAPI_TIMEDOUT`	attempt to connect ; time out without establishing the connection
`ESOCKAPI_NOMEMORY`	memory allocation error
`ESOCKAPI_INTERNALTCP`	internal error in the TCP layer
`ESOCKAPI_NOBUFS`	**datagram socket**: if the number of open datagram sockets is equal to `UPPS` (defined in *TCPIP_Stack/Configuration/tcpipcfg.h*) **stream socket**: if the number of open stream sockets is equal to `MAX_ACT_PORTS` (defined in T*CPIP_Stack/Configuration/tcpipcfg.h*

6.5 Performance

In the case of a **datagram socket** this function is a non-blocking function executed in the context of the user thread or process.

In the case of a **stream socket** this function will return at the end of the 3-way handshake protocol used by TCP to establish the connection with the remote station.

7. LISTEN()

7.1 Description

Servers that open a **stream socket** use **`listen`** to set up a queue of incoming connection requests and to open a passive connection.

Function syntax:

```
#include "sockapi.h"
int   listen( int socket,  int queuelen );
```

7.2 Inputs

`Socket`: a socket identifier created by the **socket** function

`queuelen`: number of incoming connection requests. This parameter is not used. The TCP/IP Suite sets up an infinite connection queue; the only limit is the number of available TCP descriptors for a passive connection (defined by the **`MAX_ACCEPTED_CX_ON_A_PAS_PORT`** parameter in the *TCPIP_Stack/Configuration/tcpipcfg.h* file in the TCP/IP suite).

7.3 Outputs

0 → OK

-1 → error; in this case call the **`sockapi_get_last_error`** function to get the last error

7.4 Fault messages

The returning values of the **`sockapi_get_last_error`** function are:

Error code	Description
ESOCKAPI_NOTINITIALISED	attempts to call a library function without first calling the **`sockapi_start`** function
ESOCKAPI_NOTSOCK	incorrect socket number
ESOCKAPI_ISCONN	the socket is already connected
ESOCKAPI_OPNOTSUPP	the socket type does not support **`listen`**
ESOCKAPI_NOTBOUND	the socket is not bound
ESOCKAPI_INTERNALTCP	internal error in the TCP layer
ESOCKAPI_NOMEMORY	memory allocation error
ESOCKAPI_EINTR	the blocking call was cancelled via closing of the **`ESOCKAPI_NOBUFS`** socket: if the number of open passive sockets is equal to **`MAX_PAS_PORTS`** (defined in *TCPIP_Stack/Configuration/tcpipcfg.h*).

7.5 Performance

This function is a **non-blocking function** and is executed in the context of the user thread or process.

8. ACCEPT()

8.1 Description

In the case of a **stream socket** a Server uses the **accept** function to remove the next connection request from the queue created by the **listen** function, creates a sub-socket for the request and returns the identifier of the sub-socket. The Server will interact with the Client on the new created sub-socket.

Function syntax:

```
#include "sockapi.h"
int   accept( int socket, struct sockaddr *remoteaddr, int *addrlen );
```

8.2 Inputs

`socket`: a socket identifier created by the **`socket`** function

`remoteaddr`: pointer to a **`sockaddr`** structure; this structure is filled by the **`accept`** function with the remote IP address and the remote port number;

`addrlen`: a pointer that initially specifies the size of the address structure (should be size of (**`sockaddr_in`**) and, when the call returns, specifies the number of bytes of the address structure)

8.3 Outputs

0 → OK

-1 → error; in this case call the **`sockapi_get_last_error`** function to get the last error

8.4 Fault messages

The returning values of the **`sockapi_get_last_error`** function are:

Error code	Description
ESOCKAPI_NOTINITIALISED	attempts to call a library function without first calling the **`sockapi_start`** function
ESOCKAPI_INVAL	incorrect argument (**`remoteaddr`** is 0)
ESOCKAPI_NOTSOCK	incorrect socket number
ESOCKAPI_OPNOTSUPP	the socket type does not support **`accept`**
ESOCKAPI_FAULT	incorrect address length
ESOCKAPI_NOLISTEN	**`listen`** operation was not invoked prior to **`accept`** operation
ESOCKAPI_NOBUFS	no more sub-socket descriptors are available; the maximum number that can be created is given by the **`MAX_ACCEPTED_CX_ON_A_PAS_PORT`** parameter defined in the *TCPIP_Stack/Configuration/tcpipcfg.h* file.
ESOCKAPI_NOCONREQ	the socket is non blocking and no connection requests are present in the queue to be accepted
ESOCKAPI_INTERNALTCP	internal error in the TCP layer
ESOCKAPI_EINTR	the blocking call was cancelled via closing of the socket

8.5 Performance

In the case of a **non blocking socket** if there are no connection requests in the queue then the function returns -1. A call to the function **`sockapi_get_last_error`** will return **`ESOCKAPI_NOCONREQ`**.

In the case of a **blocking socket** if there are no connection requests in the queue then the function will block the calling thread or process till the reception of a connection request from a Client. This blocking call can be cancelled by another thread that performs closes the socket that is used by **`accept`**.

9. SEND()

9.1 Description

Once an application successfully establishes a socket connection, it is ready to transfer data over the connection. An application sends data using the send function. Send is used on **stream sockets** or on **connected datagram sockets**.

When sending data care must be taken not to exceed the maximum buffer sizes of the TCP or UDP layers:

- **stream socket:** don't exceed the size defined by **`TCPSBS`** in *TCPIP_Stack/Configuration/tcpipcfg.h*
- **datagram socket:** don't exceed the size defined by (**`SIZE_OF_LRG_BUF - 52`**) in *TCPIP_Stack/Configuration/tcpipcfg.h*

In the case of a **datagram socket** if no buffer is available within the UDP layer then the function will return -1 and a call of the **`sockapi_get_last_error`** function will return **`ESOCKAPI_NOBUFS`**. This behaviour is available for both a **blocking** and a **non blocking** socket.

In the case of a **stream socket:** if no buffer is available within the TCP layer then for a:

- **blocking socket:** the function will block till the buffer contains enough space to copy the message and transmit it,
- **non blocking socket:** the function will return -1 and a call to the function **`sockapi_get_last_error`** will return **`ESOCKAPI_NOBUFS`**.

Note

The successful completion of send does not indicate that the data was successfully delivered.

Function syntax:

```
#include "sockapi.h"
int   send( int socket, char *buffer, int length, int flags );
```

9.2 Inputs

`socket`: a socket identifier created by the **socket** function

`buffer`: buffer containing the data to be transmitted

`length`: length of the data in the buffer

`flags`: not used

9.3 Outputs

If no errors occur then the function returns the total number of characters sent.

-1 → error; in this case call the function **`sockapi_get_last_error`** to get the last error.

9.4 Fault messages

The returning values of the **`sockapi_get_last_error`** function are:

Error code	Description
`ESOCKAPI_NOTINITIALISED`	attempts to call a library function without first calling the **`sockapi_start`** function
`ESOCKAPI_INVAL`	incorrect argument
`ESOCKAPI_NOTSOCK`	incorrect socket number
`ESOCKAPI_NOTCONN`	the socket is not connected
`ESOCKAPI_MSGSIZE`	the message is too large for the socket
`ESOCKAPI_NOBUFS`	no more buffers are available in TCP or UDP layer
`ESOCKAPI_CONNRESET`	the connection is closed by the remote side
`ESOCKAPI_MAXRETRANS`	give up sending the message; the message was repeated for the **`TCP_MAXRETRIES`** parameter of the TCP layer (defined in *TCPIP_Stack/Configuration/tcpipcfg.h*)
`ESOCKAPI_NOMEMORY`	memory allocation error
`ESOCKAPI_INTERNALTCP`	internal error in the TCP layer

9.5 Performance

In the case of a **non blocking socket** the function will return -1 and a call to the **`sockapi_get_last_error`** function will return **`ESOCKAPI_NOBUFS`**.

In the case of a **blocking socket** and if the socket is a **datagram socket** the function will not block the user thread or process and it will return -1 and a call to the **`sockapi_get_last_error`** function will return **`ESOCKAPI_NOBUFS`**.

In the case of a **blocking socket** and if the socket is a **stream socket** the function will block the user thread or process till the buffer contains enough space to copy the message and transmit it.

10. SENDTO()

10.1 Description

The function **`sendto`** is normally used to send a message on a **datagram socket** to a specific peer socket identified by the **`to`** parameter (see below).

On a **stream socket** the parameters **`to`** and **`tolen`** (see below) are ignored and **`sento`** is equivalent to **`send`**.

When sending data care must be taken not to exceed the maximum buffer sizes of the TCP or UDP layers:

- **stream socket:** don't exceed the size defined by **`TCPSBS`** in *TCPIP_Stack/Configuration/tcpipcfg.h*
- **datagram socket:** don't exceed the size defined by (**`SIZE_OF_LRG_BUF - 52`**) in *TCPIP_Stack/Configuration/tcpipcfg.h*

In the case of a **datagram socket** if no buffer is available within the UDP layer then the function will return -1 and a call of the **`sockapi_get_last_error`** function will return **`ESOCKAPI_NOBUFS`**. This behaviour is available for both a **blocking** and a **non blocking** socket.

In the case of a **stream socket** if no buffer is available within the TCP layer then for a:

- **blocking socket:** the function will block till the buffer contains enough space to copy the message and transmit it,
- **non blocking socket:** the function will return -1 and a call to the function **`sockapi_get_last_error`** will return **`ESOCKAPI_NOBUFS`**.

Note

The successful completion of **`sendto`** does not indicate that the data was successfully delivered.

Function syntax:

```
#include "sockapi.h"
int   sendto( int socket, char *buffer, int length, int flags, struct
sockaddr *to, int tolen );
```

10.2 Inputs

`socket`: a socket identifier created by the socket function,

`buffer`: buffer containing the data to be transmitted,

`length`: length of the data in the buffer;

`flags`: not used,

`to`: pointer to a **`sockaddr`** structure that specifies the peer IP address and the peer port number,

`tolen`: size of the address structure (should be size of (**`sockaddr_in`**)).

10.3 Outputs

If no errors occur then the function returns the total number of characters sent: -1 indicates an error. In this case call the function **`sockapi_get_last_error`** to get the last error.

10.4 Fault messages

The returning values of the function **`sockapi_get_last_error`** are:

Error code	Description
`ESOCKAPI_NOTINITIALISED`	attempts to call a library function without first calling the **`sockapi_start`** function
`ESOCKAPI_INVAL`	incorrect argument
`ESOCKAPI_FAULT`	incorrect address length
`ESOCKAPI_NOTSOCK`	incorrect socket number
`ESOCKAPI_MSGSIZE`	the message is too large for the socket
`ESOCKAPI_NOBUFS`	no more buffers are available in TCP or UDP layer
`ESOCKAPI_CONNRESET`	the connection is closed by the remote side
`ESOCKAPI_MAXRETRANS`	give up sending the message ; the message was repeated for the **`TCP_MAXRETRIES`** parameter of the TCP layer (defined in *TCPIP_Stack/Configuration/tcpipcfg.h*)
`ESOCKAPI_NOMEMORY`	memory allocation error
`ESOCKAPI_INTERNALTCP`	internal error in the TCP layer

10.5 Performance

In the case of a **non blocking socket** the function will return -1 and a call to the **`sockapi_get_last_error`** function will return **`ESOCKAPI_NOBUFS`**.

In the case of a **blocking socket** and if the socket is a **datagram socket** the function will not block the user thread or process and it will return –1. A call to the function **`sockapi_get_last_error`** will return **`ESOCKAPI_NOBUFS`**.

In the case of a **blocking socket** and if the socket is a **stream socket** the function will block the user thread or process till the buffer contains enough space to copy the message and transmit it.

11. RECV()

11.1 Description

When an application has successfully established a socket connection, it is ready to receive data over the connection. An application receives data using the **`recv`** function.

`Recv` is used on stream sockets or on connected datagram sockets.

The behaviour of this function depends on:

- the nature of the socket: **stream** or **datagram**,
- the operating mode: **blocking** or **not blocking**,
- the delivery mode: **buffered** or **not buffered**.

Stream sockets:

The TCP layer delivers the received bytes in two manners:

- **buffered delivery** permits the TCP layer to deliver the receiving bytes only when sufficient data has been received (only when the buffer contains at least **length** bytes). However if the data has arrived with the PUSH bit set (see TCP header) then the data will be delivered immediately.
- **unbuffered delivery** permits the TCP layer to deliver as much information as currently available, up to the size of **length** bytes.

In the case of a **stream blocking socket**:

- if **unbuffered delivery** is set then the function will block till the reception of a message and the size of the delivered bytes to the user is as great as currently available, up to the size of **length** bytes.
- if **buffered delivery** is set then the function will block till the TCP reception buffer contains at least **length** bytes; the size of delivered bytes to the user is **length** bytes. However if the data has arrived with the PUSH bit set (see TCP header) then the data will be delivered immediately and the size of the delivered bytes to the user is as great as is currently available, up to the size of **length** bytes.

In the case of a stream non blocking socket:

- if unbuffered delivery is set and:
  - if no bytes are received in the TCP reception buffer then the function returns -1. A call to the function **`sockapi_get_last_error`** will return **`ESOCKAPI_MSGSIZE`**.
  - the TCP reception buffer has received a message then the size of delivered bytes to the user is as great as currently available, up to the size of **length** bytes.

- if buffered delivery is set and:
  - if the number of received bytes in the TCP reception buffer contains less than **length** bytes then the function will return -1. A call to the **`sockapi_get_last_error`** function will return **`ESOCKAPI_NODATA`**.
  - if the number of received bytes in the TCP reception buffer contains at least **length** bytes then the size of delivered bytes to the user is **length** bytes. However if the data has arrived with the PUSH bit set (see TCP header) then the data will be delivered immediately and the size of the delivered bytes to the user is as great as currently available, up to the size of **length** bytes.

The default settings for a created socket are **blocking socket** and **unbuffered delivery**; these settings can be changed by a **`sockapi_ioctl`** function.

Datagram sockets:

In the case of a **datagram socket** data is extracted from the received datagram up to the size of the buffer supplied. If the datagram is larger than the buffer supplied, the buffer is filled with the first part of the datagram, the excess data is lost, and **recv** returns -1. A call to the function **`sockapi_get_last_error`** will return **`ESOCKAPI_MSGSIZE`**.

- **Datagram blocking sockets:** the function will block till the reception of a message.
- **Datagram non blocking sockets:** the function will return -1 and a call to the function **`sockapi_get_last_error`** will return **`ESOCKAPI_NODATA`**.

The default setting is **blocking socket**; this setting can be changed by a **`sockapi_ioctl`** function.

Function syntax:

```
#include "sockapi.h"
int   recv( int socket, char *buffer, int length, int flags );
```

11.2 Inputs

`socket`: a socket identifier created by the **`socket`** function,

`buffer`: buffer containing the data to be transmitted,

`length`: length of the data in the buffer,

`flags`: not used,

11.3 Outputs

If no errors: number of received bytes (see above for more details).

-1 : error; in this case call the **`sockapi_get_last_error`** function to get the last error.

11.4 Fault messages

The returning values of the **`sockapi_get_last_error`** function are:

Error code	Description
`ESOCKAPI_NOTINITIALISED`	attempts to call a library function without first calling the **`sockapi_start`** function
`ESOCKAPI_INVAL`	incorrect argument
`ESOCKAPI_NOTSOCK`	incorrect socket number
`ESOCKAPI_NOTCONN`	the socket is not connected
`ESOCKAPI_MSGSIZE`	the message is too large for the socket
`ESOCKAPI_NODATA`	the socket has no data and the socket was created as non blocking
`ESOCKAPI_CONNRESET`	the connection is closed by the remote side
`ESOCKAPI_OPFORBIDDEN`	the operation is forbidden
`ESOCKAPI_NOMEMORY`	memory allocation error
`ESOCKAPI_INTERNALTCP`	internal error in the TCP layer

11.5 Performance

In the case of a **non blocking socket** the behaviour of the function is described above.

In the case of a **blocking socket** the function will block the user thread or process. See the conditions for unblocking above.

12. RECVFROM()

12.1 Description

The function **recvfrom** is normally used to receive a message on a (possibly connected) socket from a specific peer socket identified by the **from** parameter (see below).

The behaviour of this function depends on:

- the nature of the socket: **stream** or **datagram**,
- the operating mode: **blocking** or **not blocking**,
- the delivery mode: **buffered** or **not buffered**.

For more details see **recv** function.

If the **from** parameter is not zero then this structure will be filled with the IP address and the port of the peer which sent the data.

The parameter **fromlen** should be initialised to the size of the parameter **from**, and modified on return to indicate the actual size of the address stored there.

Function syntax:

```
#include "sockapi.h"
int   recvfrom( int socket, char *buffer, int length, int flags,
struct sockaddr *from, int *fromlen);
```

12.2 Inputs

socket: a socket identifier created by the **socket** function,

buffer: buffer containing the data to be transmitted,

length: length of the data in the buffer,

flags: not used,

from: pointer to a **sockaddr** structure; **recvfrom** fills this structure with the peer IP address and the peer port number,

fromlen: size of the address structure.

12.3 Outputs

If no errors: number of received bytes (see **`recv`** function for more details)

-1: error; in this case call the **`sockapi_get_last_error`** function to get the last error.

12.4 Fault messages

The returning values of the **`sockapi_get_last_error`** function are:

Error code	Description
`ESOCKAPI_NOTINITIALISED`	attempts to call a library function without first calling the **`sockapi_start`** function
`ESOCKAPI_INVAL`	incorrect argument
`ESOCKAPI_NOTSOCK`	incorrect socket number
`ESOCKAPI_NOTCONN`	the socket is not connected
`ESOCKAPI_NOBUFS`	no more buffers are available in UDP layer
`ESOCKAPI_MSGSIZE`	the message is too large for the socket
`ESOCKAPI_NODATA`	the socket has no data and the socket was created as non blocking
`ESOCKAPI_CONNRESET`	the connection is closed by the remote side
`ESOCKAPI_OPFORBIDDEN`	the operation is forbidden
`ESOCKAPI_NOMEMORY`	memory allocation error
`ESOCKAPI_INTERNALTCP`	internal error in the TCP layer

12.5 Performance

In the case of a **non blocking socket** the behaviour of the function is described by the **`recv`** function.

In the case of a **blocking socket** the function will block the user thread or process. See **`recv`** function for the unblocking conditions.

13. CLOSESOCKET()

13.1 Description

An application calls the **`closesocket`** function after it has finished using a socket. The **`closesocket`** function frees the resources associated with the socket.

In the case of a **stream socket** the function **`closesocket`** initiates the graceful close sequence. This sequence is completed when the remote application also closes the connection. Any unread data waiting at the socket will be discarded.

Function syntax:

```
#include "sockapi.h"
int   closesocket(  int socket );
```

13.2 Inputs

`socket`: a socket identifier created by the **`socket`** function

13.3 Outputs

0 → OK

-1 → error; in this case call the **`sockapi_get_last_error`** function to get the last error.

13.4 Fault messages

The returning values of the function **`sockapi_get_last_error`** are:

Error code	Description
ESOCKAPI_NOTINITIALISED	attempts to call a library function without first calling a **`sockapi_start`** function
ESOCKAPI_NOTSOCK	incorrect socket number
ESOCKAPI_INTERNALTCP	internal error in the TCP layer

13.5 Performance

In the case of a **stream socket** the function will block the user thread or process till the end of the graceful close sequence.

In the case of a **datagram socket** the function is not blocking and is executed in the context of the user thread or process.

14. SOCKAPI_GET_LAST_ERROR()

14.1 Description

This function returns the last network error that occurred. When a particular **SockAPI** function indicates that an error has occurred, you should call this function to retrieve the appropriate error code.

Function syntax:

```
#include "sockapi.h"
int   sockapi_get_last_error();
```

14.2 Outputs

The return value indicates the error code for the last **SockAPI** function performed by this thread or process.

14.3 Performance

This function is a non blocking function and is executed in the context of the user thread or process.

15. SOCKAPI_IP2DOT()

15.1 Description

This function converts the IP address **`ip`** into a string representing the address in dotted decimal form. The user should reserve at least 18 bytes for the **`pdot`** buffer. The function will fill the **`pdot`** buffer.

Function syntax:

```
#include "sockapi.h"
void  sockapi_ip2dot( char *pdot, IPAddr ip );
```

15.2 Inputs

`pdot`: address of a buffer; this buffer is allocated by the user and will be filled by **`sockapi_ip2dot`.**

`ip`: IP address to be converted.

15.3 Performance

This function is a non blocking function and is executed in the context of the user thread or process.

16. SOCKAPI_DOT2IP()

16.1 Description

This function converts the string **`pdot`** representing the IP address in dotted decimal form into an **`IPaddr`** address.

Function syntax:

```
#include "sockapi.h"
void  sockapi_dot2ip( IPAddr ip, char *pdot );
```

16.2 Inputs

`ip`: IP address filled by **`sockapi_dot2ip`.**

`pdot`: the string to be converted, containing the IP address in dotted decimal form.

16.3 Performance

This function is a non blocking function executed in the context of the user thread or process.

Chapter 9 TCP/IP Stack module functions

1. TCPIP_GETADDR(): REQUESTING THE IP ADDRESS OF A LOCAL INTERFACE

1.1 Description

If the point-to-point IP address of an interface is not known, the user can request this address using the **`tcpip_getaddr()`** function. In this case, a **RARP server** must be present on the remote machine.

The invocation of the **`tcpip_getaddr()`** function should be performed after calling the **`tcpip_start()`** function; the RARP mechanism will be initiated.

Function syntax:

```
#include "sockapi.h"
tcpip_getaddr(unsigned short if_number, IPaddr ip_addr);
```

1.2 Input parameters

`if_number`: interface number; this number is obtained following the call to **`tcpip_start()`**.

`ip_addr`: IPaddr type array; all bytes must be set to 0.

1.3 Output parameters

`ip_addr`: the RARP protocol is launched and the IP interface address is copied into the **`ip_addr`** table by the TCP/IP Stack module.

1.4 Returned values

Value	Description
0	OK
1	if the requested interface is not initialised, then it needs to be correctly initialised following the **`tcpip_start`** request
2	if there is a system error
3	if the interface number is incorrect
4	if there is no response to the RARP request after **`ARP_MAXRETRY`** retransmissions (defined in the *TCPIP_Stack/Configuration/tcpipcfg.h* file).
5	if the TCP/IP stack has been stopped by **`tcpip_stop()`**

2. TCPIP_PUTADDR(): SETTING THE IP ADDRESS OF A LOCAL INTERFACE

2.1 Description

If the point-to-point IP address of an interface is not known and if there is no RARP server on the network, then the user can obtain his IP address by implementing an application that makes use of the **Bootstrap** or another protocol. He gets his address with an application that uses UDP and the "limited broadcast" IP address (255.255.255.255).

The default interface that accepts "limited broadcast" is interface number 1. If a workstation is configured as a host, there is no problem as there is only one interface; if the workstation is configured as a router (several interfaces), only interface no. 1 accepts this broadcasting mode.

Once this address is obtained, the user must enter it into the **TCP/IP Stack** module using the **`tcpip_putaddr()`** function.

Function syntax:

```
#include "sockapi.h"
void  tcpip_putaddr(unsigned short if_number,  IPaddr ip_addr);
```

2.2 Input parameters

`if_number`: interface number, must be 1.

`ip_addr`: IPaddr type array.

2.3 Output parameters

None.

3. TCPIP_CONF_TARP(): STATIC CONFIGURATION OF THE IP ADDRESS AND HARDWARE ADDRESS CORRESPONDENCE TABLE

3.1 Description

The invocation of the **`tcpip_conf_tarp()`** function is optional; if the user makes this call, then the IP address and hardware address correspondence table will be filled in statically. This is a way of replacing the ARP mechanism.

In any case, the ARP mechanism works for any IPx address whose <**`IPx Addr.`**><**`hardx Addr.`**> couple does not exist in the IP address and hardware address correspondence table.

Function syntax:

```
#include "sockapi.h"
tcpip_conf_tarp(unsigned short if_number,
      unsigned short nb_of_couples,

IP_HARD_COUPLE **ip_hard_table);
```

3.2 Input parameters

`if_number`: interface number.

`nb_of_couples`: number of <**`IP Addr.><Hard Addr.`**> couples in the table pointed to by **`ip_hard_table`**.

`ip_hard_table`: pointer to a table of **`nb_of_couples`** pointers to **`IP_HARD_COUPLE`** type structures, as defined below.

`ip_addr`: IP address of a remote workstation.

`hw_addr`: pointer to a 3-byte (WorldFIP) or 6-byte (Ethernet) array, providing the hardware address of the remote workstation.

```
typedef struct {
        Ipaddr              ip_addr;
        unsigned char       *hw_addr;
}IP_HARD_COUPLE;
```

3.3 Output parameters

None.

3.4 Returned values

Value	Description
0	OK
1	if the requested interface is not initialised, then it needs to be correctly initialised by the **`tcpip_start()`** function.
2	if there is a system error.
3	if the interface number is incorrect
4	if the **`nb_of_couples`** parameter is greater than **`ARP_TSIZE`** (defined in the *TCPIP_Stack/Configuration/tcpipcfg.h* file).

3.5 Example

Let us suppose that there are 1 Ethernet interface and 2 ALSTOM WorldFIP interfaces (see the example used for the description of the utilisation of the **`tcpip_start()`**function).

- **WorldFIP1 interface** is connected on a WorldFIP network segment that has 3 stations connected (itself + 2 other stations),
- **WorldFIP2 interface** is connected on a WorldFIP network segment that has 3 stations connected (itself + 2 other stations),
- **Ethernet interface** is connected on a Ethernet network segment that has 2 stations connected (itself + 1 other station).

```
                    /* IP and hardware addresses of the stations on the Ethernet network */

#define IP_ST1_ETH          "192.9.230.1"          /* local */
#define IP_ST2_ETH          "192.9.230.2"          /* remote */


static unsigned char    eth_addr_st1[] = {0x8, 0, 0x20, 0x1f, 0x44, 0x67}; /* local */
static unsigned char    eth_addr_st2[] = {0x8, 0, 0x20, 4, 0xf, 0x37}; /* remote */

static IP_HARD_COUPLE       ip_eth_st1 = {
                                        {0, 0, 0, 0},          /* IP addr */
                                        eth_addr_st1
                                    };

static IP_HARD_COUPLE       ip_eth_st2      = {
                                        {0, 0, 0, 0},          /* IP addr */
                                        eth_addr_st2
                                    };

                    /* IP and hardware addresses of the stations on the WorldFIP1 network */

#define IP_ST1_FIP1         "192.9.250.1"   /* local */
#define IP_ST2_FIP1         "192.9.250.2"   /* remote */
#define IP_ST4_FIP1         "192.9.250.4"   /* remote  */
```

```
                                        /*LSAP, Station, Segment */
static unsigned char    addr_st1_fip1[] = {0x01,  0x01,  0x0};  /* local */
static unsigned char    addr_st2_fip1[] = {0x01,  0x02,  0x0};  /* remote */
static unsigned char    addr_st4_fip1[] = {0x01,  0x04,  0x0};  /* remote */


static IP_HARD_COUPLE       ip_st2_fip1     = {
                                    {0, 0, 0, 0},           /* IP addr */
                                    addr_st2_fip1
                                };

static IP_HARD_COUPLE       ip_st4_fip1     = {
                                    {0, 0, 0, 0},           /* IP addr */
                                    addr_st4_fip1
                                };

          /* IP and hardware addresses of the stations on the WorldFIP2 network */

#define IP_ST3_FIP2             "192.9.254.31" /* local */
#define IP_ST5_FIP2             "192.9.254.5"       /* remote */
#define IP_ST7_FIP2             "192.9.254.7"       /* remote */

                                        /*LSAP, Station, Segment */
static unsigned char    addr_st3_fip2[] = {0x01,  0x1F,  0x01};/* local */
static unsigned char    addr_st5_fip2[] = {0x01,  0x05,  0x01};     /* remote */
static unsigned char    addr_st7_fip2[] = {0x01,  0x07,  0x01};     /* remote */


static IP_HARD_COUPLE       ip_st5_fip2     = {
                                    {0, 0, 0, 0},           /* IP addr */
                                    addr_st5_fip2
                                };

static IP_HARD_COUPLE       ip_st7_fip2     = {
                                    {0, 0, 0, 0},           /* IP addr */
                                    addr_st7_fip2
                                };

                    /* tables */

static    IP_HARD_COUPLE    *ip_eth_table[1] = {
                                        &ip_eth_st2
                                    };

static    IP_HARD_COUPLE    *ip_fip1_table[2] = {
                                        &ip_st2_fip1,
                                        &ip_st4_fip1
                                    };

static    IP_HARD_COUPLE    *ip_fip2_table[2] = {
                                        &ip_st5_fip2,
                                        &ip_st7_fip2
                                    };
```

```
user_program()
{
        ..........................................
        int       rv;

        ..................................................................
         sockapi_dot2ip(ip_st2_eth.ip_addr, IP_ST2_ETH);
         sockapi_dot2ip(ip_st2_fip1.ip_addr, IP_ST2_FIP1);
         sockapi_dot2ip(ip_st4_fip1.ip_addr, IP_ST4_FIP1);
         sockapi_dot2ip(ip_st5_fip2.ip_addr, IP_ST5_FIP2);
         sockapi_dot2ip(ip_st7_fip2.ip_addr, IP_ST7_FIP2);

           rv = tcpip_conf_tarp(eth_if_no, 1, ip_eth_table);
           if(rv != 0)
              printf("\n tcpip_conf_tarp : Error = %d\n", rv);

           rv = tcpip_conf_tarp(fip1_if_no, 2, ip_fip1_table);
           if(rv != 0)
              printf("\n tcpip_conf_tarp : Error = %d\n", rv);

           rv = tcpip_conf_tarp(fip2_if_no, 2, ip_fip2_table);
           if(rv != 0)
              printf("\n tcpip_conf_tarp : Error = %d\n", rv);

        ..............................................................

}
```

4. TCPIP_RTADD(): ADDING AN ELEMENT TO THE ROUTING TABLE

4.1 Description

If the user has implemented protocols such as RIP or OSPF above the **TCPIP stack** module, then he can add a "route" element to the routing table by invoking the **`tcpip_rtadd()`** function. Otherwise, the user must invoke this function only if he is certain of the data being entered.

Function syntax:

```
#include "sockapi.h"

tcpip_rtadd( IPaddr rt_net,
             IPaddr rt_mask,
             IPaddr rt_gw,
             int rt_metric,
             unsigned short rt_ifnum,
             int rt_ttl);
```

4.2 Input parameters

`rt_net`: IP address that must contain the network address for this route.

`rt_mask`: address that must contain the mask for this route.

`rt_gw`: IP address that must contain the IP address of the next "hop".

`rt_metric`: distance expressed as a number of "hops".

`rt_ifnum`: interface number to which the message is transmitted.

`rt_ttl`: time, in seconds, required to validate the existence of the "route" element in the routing table; when this field reaches the value 0, this element is removed from the table. The user can set an infinite time for this element (**`RT_INF`** value defined in the *TCPIP_Stack/Basic_Stack/route.h* file), thus rendering this element permanent.

4.3 Output parameters

None.

4.4 Returned values

Value	Description
0	OK
1	if the requested interface is not initialised, then it needs to be correctly initialised by the **`tcpip_start()`** function.
2	if there is a system error.

5. TCPIP_RTDEL(): DELETING AN ELEMENT FROM THE ROUTING TABLE

5.1 Description

Deletion of a "route" element (added by **`tcpip_rtadd()`**) from the routing table is performed by the user through invocation of the **`tcpip_rtdel()`** function.

Function syntax:

```
#include "sockapi.h"

tcpip_rtdel( IPaddr rt_net,
      IPaddr rt_mask);
```

5.2 Input parameters

`rt_net`: IP address that must contain the network address for this route.

`rt_mask`: IP address that must contain the mask for this route.

5.3 Output parameters

None.

5.4 Returned values

Value	Description
0	OK
1	if the requested interface is not initialised, then it needs to be correctly initialised by the **`tcpip_start()`** function.
2	if there is a system error

6. TCPIP_ECHO_RQ(): SENDING AN ECHO REQUEST

6.1 Description

To test whether a device is present, the user can query it by invoking the **`tcp_echo_rq()`** function. This call addresses the **ICMP** layer inside the **TCP/IP Stack** module, which builds and issues the **`ECHO`** (**`ICMP`** type) message.

After the reception of the ECHO response, the **TCP/IP Stack** module calls the user callback function whose address is:

- **`addr_user_echo_reply`**: this address was set inside the TCP/IP Stack module during the call to **`tcpip_start()`** function;

  or

- **`user_echo_resp`**: this address was set inside the TCP/IP Stack module during the call to **`tcpip_echo_rq()`** function;

In a mono-tasking environment, or in a multi-tasking environment if the user requires a unique ECHO response processing function, the user must define the **`addr_user_echo_reply`** parameter when the **`tcpip_start()`** function is called. Otherwise this parameter must be null.

In a multi-tasking environment if the user requires a task- (or process-) based ECHO response processing function, then he must define the **`user_echo_reply`** parameter when the **`tcpip_echo_rq()`** function is called. Otherwise this parameter must be null.

Function syntax:

```
#include "sockapi.h"

tcpip_echo_rq(    IPaddr dst,
     unsigned short  pid,
     unsigned short  seq_nb,  int data_len,
     void (*user_echo_reply)(struct ip *pip) );

void user_echo_reply( struct ip *pip );
```

6.2 Input parameters

`dst`: destination IP address

`pid`: calling process (or task) identifier.

`seq_nb`: sequence number enabling the user to match the request with the response.

`data_len`: length of additional data to transmit in the echo message; must be an even number.

`user_echo_reply`: address of a user function that will be called following an ECHO response; in this case the **`addr_user_echo_reply`** is null when the **`tcpip_start()`** function is called.

6.3 Output parameters

None.

6.4 Returned values

Value	Description
0	OK
1	if the requested interface is not initialised, then it needs to be correctly initialised by the function **`tcpip_start()`**.
2	if there is a system error
3	if the TCP/IP stack cannot find a route to the requested IP address.
4	if the **`user_echo_resp`** parameter is not null and the address of the ECHO response processing function is already defined (the **`adr_ser_echo_resp`** parameter is non-null at the time of the **`tcpip_start()`**request).
5	if the number of processes (tasks), identified by their **`pid`** and making the ECHO request, is greater than the **`IC_MAX_ECHO_PROC`** parameter (defined in the *TCPIP_Stack/Configuration/tcpipcfg.h* file).

6.5 Example

```
#define IP_ST2_ETH              "192.9.230.2"    /* distant */

user_program()
{
        IPaddr                  ip_addr_st2;;
        unsigned short          pid = 678;
        int                     seq_nb;
        int                     data_len;
        ..........................................................
        sockapi_dot2ip(ip_addr_st2, IP_ST2_ETH);
        seq_nb = 1;
        data_len = 20;
        rv = tcpip_echo_rq(ip_addr_st2, pid, seq_nb, data_len, 0);
        ..........................................................
}


static void user_echo_reply(struct ip *pip)
{
        short                   len;
        unsigned char           icmp_buff[300];
        short                   i;

                /* print the ECHO response */

        len = pip->ip_len;

        /* copy the bytes pointed by pip in icmp_buff */

        fprintf(fd, " ECHO Reponse = \n");
        for(i=0; i < len; i++)
          printf("%x ",  icmp_buff[i] );
        printf("\n");
}
```

7. TCPIP_GET_SUBNETMASK(): GETTING THE SUBNET MASK

7.1 Description

To get the subnet mask, the user must invoke the **`tcpip_get_subnetmask()`** function. This call addresses the ICMP layer inside the **TCP/IP Stack** module, which builds and broadcasts a mask request message.

Following receipt of the response, the **TCP/IP Stack** module calls a user callback function whose address was communicated to the TCP/IP Stack module by the parameter **`adr_user_mask_reply`** when the **`tcpip_start()`** function was called.

Function syntax:

```
#include "sockapi.h"
tcpip_get_subnet_mask( unsigned short if_number);
void user_mask_reply( struct ip *pip );
```

7.2 Input parameters

`if_number`: interface number.

7.3 Output parameters

None.

7.4 Returned Values

Value	Description
0	OK
1	if the requested interface is not initialised, then it needs to be correctly initialised by the function **`tcpip_start()`**.
2	if there is a system error

7.5 Example

```
user_program()
{
        .....

        rv = tcpip_get_subnetmask(1);
        .....
}



static void user_mask_reply(struct ip *pip)
{
        struct icmp             *picmp;
        unsigned char           subnet_mask[4];
        short                   i;

        picmp = (struct icmp *)pip->ip_data;

        memcpy( (char *)subnet_mask, ( char *)picmp->ic_data, 4 );

        fprintf(fd, "Subnet mask = ");
        for(i=0; i < 4; i++)
          printf("%x ",  subnet_mask[i] );
        printf("\n");
}
```

Chapter 10 Processing the real-time clock interrupt

1.1 Introduction

The hardware environment should have a programmable timer that periodically produces interrupts. You should connect the interrupt vector to the interrupt handler. This operation depends on the user hardware environment and operating system.

The user has to call the **`tbx_TimerUpToDate()`** function inside the body of the interrupt handler.

The period for calling the **`tbx_TimerUpToDate()`** function must correspond to the value **`TIMER_TICK`** set by the user in the file *TCPIP_Stack/Configuration/tcpipcfg.h***.**

1.2 Example

```
handler_timer()
{
……
   tbx_TimerUpToDate();
……

}
```

Appendix A

Example of a stream socket client

```
#include "sockapi.h"


#define ECHO_PORT                                   7
#define DAYTIME_PORT                        13
#define TIME_PORT                           37
#define HTTP_PORT                           80

main()
{
.....
int             rv;
   int                  error;
    FILE                *fd = stderr;
.....

                        /* launch the TCP/IP suite */
    rv = tcpip_start(0, 0, &if_eth, &signals_addr, &icmp_resp_addr);

    if(rv != 0){
      fprintf(fd, "\n tcpip_start : Error = %d\n", rv);
    }
.....
/* launch the SockAPI library */
     if(sockapi_start(0) != 0){
            error = sockapi_get_last_error();
            fprintf(fd, "\n sockapi_start : Error = %d\n", error);
     }
.....

stream_clients();               /* example of stream clients */
stream_server();                /* example of stream server */
datagram_clients();   /* example of datagram clients */
datagram_server();    /* example of datagram server */
.....
/* stop the SockAPI library */
sockapi_stop();
.....

                        /* stop the TCP/IP suite */
tcpip_stop();
.....
}
```

This example of a stream socket client opens 3 clients:

- on socket 1, a blocking ECHO client
- on socket 2, a blocking DAYTIME client
- on socket 3, a non blocking TIME client

```
static void stream_clients()
{
    int                         sock1 = 0;
    int                         sock2 = 0;
    int                         sock3 = 0;
    struct sockaddr_inlocaladdr1, localaddr2, localaddr3;
    struct sockaddr_inremoteaddr1, remoteaddr2, remoteaddr3;
    struct sockaddr_into, from;
    IPaddr                      rip, ripfrom;
    int                         fromlen = sizeof( struct sockaddr_in );
    int                         error;
    char                        msg[300];
    char                        rcv_buff[310];
    int                         rv;
    int                         i,j;


            /***** DAYTIME CLIENT - blocking client ****/

                                /* open the socket 2 - DAYTIME */

        sock2 = socket(PF_INET, SOCK_STREAM, 0);
        if(sock2 == -1){
                error = sockapi_get_last_error();
                fprintf(fd, "\n SOCK2 socket : Error = %d\n", error);
                goto endclient;
        }
                                        /* bind to a local endpoint */

        localaddr2.sin_family = AF_INET;
        localaddr2.sin_addr.s_addr = INADDR_ANY;
        localaddr2.sin_port = 1000 + DAYTIME_PORT;

        rv = bind(sock2, (struct sockaddr*)&localaddr2,
                        sizeof(localaddr2) );
         if(rv == -1){
                error = sockapi_get_last_error();
                fprintf(fd, "\n SOCK2 bind : Error = %d\n", error);
                goto endclient;
        }
                                        /* connect the socket */

         remoteaddr2.sin_family = AF_INET;
         sockapi_dot2ip(rip, IP_ST2_ETH);
         memcpy((unsigned char*)&remoteaddr2.sin_addr,
                (unsigned char *)rip, IP_ADDLEN);
         remoteaddr2.sin_port = DAYTIME_PORT;

         rv = connect(sock2, (struct sockaddr *)&remoteaddr2,
                                sizeof(remoteaddr2));
         if(rv == -1){
                error = sockapi_get_last_error();
                fprintf(fd, "\n SOCK2 connect : Error = %d\n", error);
```

```
        goto endclient;
}
                /* Receive on the DAYTIME Socket */

rv = recv(sock2, rcv_buff, 100, 0);
if(rv == -1){
        error = sockapi_get_last_error();
        fprintf(fd, "\n SOCK2 recv : Error = %d\n", error);
        goto endclient;
}
else{
        rcv_buff[rv] = '\0';
        fprintf(fd, "\n DAYTIME : ");
        fprintf(fd, "%s", rcv_buff);
        fprintf(fd, "\n\n");
}
                        /* close the connection */
if(sock2>0){
        rv = closesocket(sock2);
        if(rv == -1){
           error = sockapi_get_last_error();
           fprintf(fd, "\n SOCK2 close socket : Error = %d\n",error);
        }
        sock2 = 0;
}


        /***** TIME CLIENT - non blocking client ****/

                        /* open the socket 3 TIME */

sock3 = socket(PF_INET, SOCK_STREAM, 0);
if(sock3 == -1){
        error = sockapi_get_last_error();
        fprintf(fd, "\n SOCK3 socket : Error = %d\n", error);
        goto endclient;
}
                                /* no bind */

                                /* set as non blocking */

 rv = sockapi_ioctl(sock3, FIONBIO, (char *)1);
 if(rv == -1){
        error = sockapi_get_last_error();
        fprintf(fd, "\n SOCK3 ioctl : Error = %d\n", error);
        goto endclient;
}
                                /* connect the socket */

 remoteaddr3.sin_family = AF_INET;
 sockapi_dot2ip(rip, IP_ST2_ETH);
 memcpy((unsigned char*)&remoteaddr3.sin_addr,
           (unsigned char *)rip, IP_ADDLEN);
 remoteaddr3.sin_port = TIME_PORT;

 rv = connect(sock3, (struct sockaddr *)&remoteaddr3,
                        sizeof(remoteaddr3));
 if(rv == -1){
        error = sockapi_get_last_error();
        fprintf(fd, "\n SOCK3 connect : Error = %d\n", error);
        goto endclient;
```

```
}
                    /* Receive on the TIME Socket */

rv = -1;
k = 0;
while(rv = = -1){
        rv = recv(sock3, rcv_buff, 100, 0);
        if(rv = = -1){
           error = sockapi_get_last_error();
           if(error = = ESOCKAPI_NODATA){
                    if(k % STAR = = 0)
                       fprintf(fd,"*");
                    k + +;
           }
           else{
                    fprintf(fd, "\n SOCK3 recv : Error = %d\n", error);
                    goto endclient;
           }
        }
        else{
           fprintf(fd, "\n\n TIME : ");
           for(i=0; i<rv; i+ +)
                    fprintf(fd, "%x ", rcv_buff[i]);
           fprintf(fd, "\n\n");
        }
}
                                /* close the connection */
if(sock3>0){
        rv = closesocket(sock3);
        if(rv = = -1){
           error = sockapi_get_last_error();
           fprintf(fd, "\n SOCK3 close socket : Error = %d\n",error);
        }
        sock3 = 0;
}


        /***** ECHO CLIENT - blocking client ****/

                                /* open the socket 1 - ECHO */

sock1 = socket(PF_INET, SOCK_STREAM, 0);
if(sock1 = = -1){
        error = sockapi_get_last_error();
        fprintf(fd, "\n SOCK1 socket : Error = %d\n", error);
        goto endclient;
}

                                        /* bind to any local endpoint */

localaddr1.sin_family = AF_INET;
localaddr1.sin_addr.s_addr = INADDR_ANY;
localaddr1.sin_port = 0;                                /* any port */

rv = bind(sock1, (struct sockaddr*)&localaddr1,
                        sizeof(localaddr1) );
 if(rv = = -1){
        error = sockapi_get_last_error();
        fprintf(fd, "\n SOCK1 bind : Error = %d\n", error);
        goto endclient;
}
```

```
                                        /* connect the socket */

 remoteaddr1.sin_family = AF_INET;
 sockapi_dot2ip(rip, IP_ST2_ETH);
 memcpy((unsigned char*)&remoteaddr1.sin_addr,
          (unsigned char *)rip, IP_ADDLEN);
 remoteaddr1.sin_port = ECHO_PORT;

 rv = connect(sock1, (struct sockaddr *)&remoteaddr1,
                              sizeof(remoteaddr1));
 if(rv == -1){
        error = sockapi_get_last_error();
        fprintf(fd, "\n SOCK1 connect : Error = %d\n", error);
        goto endclient;
}
                /* Send and Receive on the ECHO socket
                   using send and recv functions */

for(j = 0; j < 32; j++){

  for(i=0; i<100; msg[i] = j, i++);
  rv = send(sock1, msg, 100, 0);
  if(rv == -1){
        error = sockapi_get_last_error();
        fprintf(fd, "\n SOCK1 send : Error = %d\n", error);
        goto endclient;
  }

  rv = recv(sock1, rcv_buff, 100, 0);
  if(rv == -1){
        error = sockapi_get_last_error();
        fprintf(fd, "\n SOCK1 recv : Error = %d\n", error);
        goto endclient;
  }
  else{
        fprintf(fd, "\n ECHO : Received message = ");
        for(i=0; i<10; i++)
            fprintf(fd, "%x ", rcv_buff[i]);
        fprintf(fd, "\n\n");

  }
}
                /* Send and Receive on the ECHO socket
                   using sendto and recvfrom functions */

memcpy((char *)&to, (char *)&remoteaddr1,
                                        sizeof(struct sockaddr_in));
for(j = 32; j < 64; j++){

  for(i=0; i<100; msg[i] = j, i++);
  rv = sendto(sock1, msg, 100, 0,
                (struct sockaddr *)&to, sizeof(to));
  if(rv == -1){
        error = sockapi_get_last_error();
        fprintf(fd, "\n SOCK1 send : Error = %d\n", error);
        goto endclient;
  }

  rv = recvfrom(sock1, rcv_buff, 200, 0,
                (struct sockaddr *)&from, &fromlen);
```

```
                if(rv == -1){
                        error = sockapi_get_last_error();
                        fprintf(fd, "\n SOCK1 recv : Error = %d\n", error);
                        goto endclient;
                }
                else{
                        memcpy((char *)ripfrom, (char *)&from.sin_addr, IP_ADDLEN);
                        fprintf(fd, "\n ECHO : Received message = ");
                        for(i=0; i<10; i++)
                            fprintf(fd, "%x ", rcv_buff[i]);
                        fprintf(fd, "\n        FROM remote station ");
                        for(i=0; i<IP_ADDLEN; i++)
                            fprintf(fd, "%d.", ripfrom[i]);
                        fprintf(fd, "\n\n");
                }
        }
                                        /* close the connection */
        if(sock1>0){
                rv = closesocket(sock1);
                if(rv == -1){
                   error = sockapi_get_last_error();
                   fprintf(fd, "\n SOCK1 close socket : Error = %d\n",error);
                }
                sock1 = 0;
        }


endclient :
        if(sock1>0){
                rv = closesocket(sock1);
                if(rv == -1){
                   error = sockapi_get_last_error();
                   fprintf(fd, "\n SOCK1 close socket : Error = %d\n",error);
                }
                sock1 = 0;
        }

        if(sock2>0){
                rv = closesocket(sock2);
                if(rv == -1){
                   error = sockapi_get_last_error();
                   fprintf(fd, "\n SOCK2 close socket : Error = %d\n",error);
                }
                sock2 = 0;
        }

        if(sock3>0){
                rv = closesocket(sock3);
                if(rv == -1){
                   error = sockapi_get_last_error();
                   fprintf(fd, "\n SOCK3 close socket : Error = %d\n",error);
                }
                sock3 = 0;
        }
    }
}
```

Appendix B

Example of a stream socket server

This example shows how to implement a simple HTTP server that increments a counter each time a browser opens a connection with it.

```
static char header[] = "HTTP/1.0 200 OK \r\n\
Server: Embedded 1.0\r\n\
Content-Type: text/html\r\n\r\n";

char response_page[]="<HTML><BODY> <H1> Count : %d </H1></BODY></HTML>";

static char buffer[512];
static counter_page;

stream_server()
{
    server_loop(HTTP_PORT);
}


void server_loop(unsigned short service)
{
 char                       Buffer[1024];
 int                        msock, ssock;
 int                        alen,i ;
 struct sockaddr_in    fsin;
 int                        stat;
 int                   last_err;

 counter_page = 1;
 msock = passivesocketTCP(service,10);
 if(msock<0)
  {
    fprintf(fd,"\nsocket on main sock failed,code: %d\n",sockapi_get_last_error());
    return;
  }
 else
   fprintf(fd,"\npassive socket created on port %d\n",service);

  continuer_server = 1;
  while(continuer_server == 1){

     alen = sizeof(fsin);
     ssock =accept(msock,(struct sockaddr*)&fsin, &alen);
```

```
        if(ssock<0){
                fprintf(fd,"\n accept failed,code: %d\n",sockapi_get_last_error());
                goto endsrv;;
        }

         for(i=0;i<Size;i++) Buffer[i] =(char)0;

        stat = recv (ssock, Buffer, Size,0);

           if(stat>0){

                fprintf(fd,"\ncorrect reception\n%s\n....\n ",Buffer);
                sprintf(buffer,response_page,counter_page);
                fprintf(fd,response_page,counter_page);

                stat = send (ssock, header, strlen(header), 0);
                if(stat<0)
                  fprintf(fd,"\nsend bad status,code: %d\n",
                                                sockapi_get_last_error());
                else
                   fprintf(fd,"send OK\n");

                stat = send (ssock,buffer,strlen(buffer),0);
                if(stat<0)
                  fprintf(fd,"\nsend bad status,code: %d\n",
                                                        sockapi_get_last_error());
                else
                   fprintf(fd,"\nsend OK\n");

           }else
                fprintf(fd,"\n reception failed\n");

        stat = closesocket(ssock);
        counter_page++;
        if(stat<0) {
           last_err = sockapi_get_last_error();
           if(last_err != ESOCKAPI_CONNRESET){
              fprintf(fd,"\nbad close sub-socket status,code: %d\n",last_err);
              goto endsrv;
           }
        }
        else
           fprintf(fd,"\nsub-socket closesocket OK\n");
      }
    endsrv :

        stat = closesocket(msock);
        if(stat<0)
                fprintf(fd,"\nbad close main socket status , code: %d\n",
    sockapi_get_last_error());
        else
                fprintf(fd,"\n main socket closesocket OK\n");

        fprintf(fd,"\n !!!! END OF SERVER \n");

    }



    int         passivesocketTCP( unsigned short service, int qlen)
    {
```

```
return passivesocket( service, "tcp",qlen);
}



static int    passivesocket(u_short service, char* protocol, int qlen)
{
    struct sockaddr_in sin;       /* an Internet endpoint address        */
    int s,type;                             /*socket descriptor, socket type        */
    int i;

    bzero((char*)&sin,sizeof(sin));

    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;
    sin.sin_port = service;

    if (strcmp(protocol,"udp")==0)
              type = SOCK_DGRAM;
    else
              type =SOCK_STREAM;

              /* allocate socket */
    s = socket (PF_INET,type,0);

    if (s<0) {
              fprintf(fd,"cant create socket, reason: %d\n",
                                                  sockapi_get_last_error());
              exit(1);
   };

     i = bind(s,(struct sockaddr*)&sin,sizeof(sin));


    if (i<0){
              fprintf(fd,"can't bind the socket to port %d ---reason: %d\n",
              service, sockapi_get_last_error());
              exit(1);
   };

    if(type == SOCK_STREAM && (listen(s,qlen)<0)){
              fprintf(fd,"can't listen the socket to port %d --- %s\n",
              service, sockapi_get_last_error());
              exit(1);
   };

    return s;
}
```

Appendix C

Example of a datagram socket client

This example opens three clients:

- on socket 1, a non blocking and non connected ECHO client 1,
- on socket 2, a blocking ECHO client 2 connected with bind,
- on socket 3, a blocking ECHO client 3 connected without bind.

```
static void datagram_clients()
{
    int                         sock1 = 0;
    int                         sock2 = 0;
    int                         sock3 = 0;
    struct sockaddr_inlocaladdr1, localaddr2, localaddr3;
    struct sockaddr_inremoteaddr1, remoteaddr2, remoteaddr3;
    struct sockaddr_infrom;
    Ipaddr                      rip, ripfrom;
    int                         fromlen = sizeof( struct sockaddr_in );
    int                         error;
    char                        msg[300];
    char                        rcv_buff[310];
    int                         rv;
    int                         i,j,k;

/***** ECHO CLIENT 1 :
                                - non blocking,
- non connected datagram****/

        sock1 = socket(PF_INET, SOCK_DGRAM, 0);
        if(sock1 == -1){
                error = sockapi_get_last_error();
                fprintf(fd, "\n SOCK1 socket : Error = %d\n", error);
                goto endclient;
        }
                                        /* set as non blocking */

         rv = sockapi_ioctl(sock1, FIONBIO, (char *)1);
         if(rv == -1){
                error = sockapi_get_last_error();
                fprintf(fd, "\n SOCK1 ioctl : Error = %d\n", error);
                goto endclient;
        }
                                        /* bind to any local endpoint */

        localaddr1.sin_family = AF_INET;
```

```
localaddr1.sin_addr.s_addr = INADDR_ANY;
localaddr1.sin_port = 0;                                /* any port */

rv = bind(sock1, (struct sockaddr*)&localaddr1,
                        sizeof(localaddr1) );
 if(rv == -1){
        error = sockapi_get_last_error();
        fprintf(fd, "\n SOCK1 bind : Error = %d\n", error);
        goto endclient;
}

 remoteaddr1.sin_family = AF_INET;
 sockapi_dot2ip(rip, IP_ST2_ETH);
 memcpy((unsigned char*)&remoteaddr1.sin_addr,
            (unsigned char *)rip, IP_ADDLEN);
 remoteaddr1.sin_port = ECHO_PORT;

                                        /* Send and Receive */

for(j = 0; j < 16; j++){

  for(i=0; i<100; msg[i] = j, i++);
  rv = sendto(sock1, msg, 100, 0,
                    (struct sockaddr *)&remoteaddr1, sizeof(remoteaddr1));
  if(rv == -1){
        error = sockapi_get_last_error();
        fprintf(fd, "\n SOCK1 send : Error = %d\n", error);
        goto endclient;
  }

  rv = -1;
  k = 0;
  while(rv == -1){
    rv = recvfrom(sock1, rcv_buff, 200, 0,
                    (struct sockaddr *)&from, &fromlen);
        if(rv == -1){
            error = sockapi_get_last_error();
            if(error == ESOCKAPI_NODATA){
                    if(k % STAR == 0)
                       fprintf(fd,"*");
                    k++;
            }
            else{
                    fprintf(fd, "\n SOCK1 recv : Error = %d\n", error);
                    goto endclient;
            }
        }
        else{
            memcpy((char *)ripfrom, (char *)&from.sin_addr, IP_ADDLEN);
            fprintf(fd, "\n ECHO : Received message = ");
            for(i=0; i<10; i++)
                    fprintf(fd, "%x ", rcv_buff[i]);
                    fprintf(fd, "\n       FROM remote station ");
                    for(i=0; i<IP_ADDLEN; i++)
                       fprintf(fd, "%d.", ripfrom[i]);
                 fprintf(fd, "\n\n");
        }
  }

}
                                /* close the connection */
```

```
                if(sock1>0){
                        rv = closesocket(sock1);
                        if(rv == -1){
                           error = sockapi_get_last_error();
                           fprintf(fd, "\n SOCK1 close socket : Error = %d\n",error);
                        }
                        sock1 = 0;
                }

                        /***** ECHO CLIENT 2 :
        - blocking,
        - connected datagram with bind ****/

                sock2 = socket(PF_INET, SOCK_DGRAM, 0);
                if(sock2 == -1){
                        error = sockapi_get_last_error();
                        fprintf(fd, "\n SOCK2 socket : Error = %d\n", error);
                        goto endclient;
                }
                                                /* bind to a local endpoint */

                localaddr2.sin_family = AF_INET;
                localaddr2.sin_addr.s_addr = INADDR_ANY;
                localaddr2.sin_port = 3000 + ECHO_PORT;

                rv = bind(sock2, (struct sockaddr*)&localaddr2,
                                sizeof(localaddr2) );
                 if(rv == -1){
                        error = sockapi_get_last_error();
                        fprintf(fd, "\n SOCK2 bind : Error = %d\n", error);
                        goto endclient;
                }

                                                /* connect the socket */

                 remoteaddr2.sin_family = AF_INET;
                 sockapi_dot2ip(rip, IP_ST2_ETH);
                 memcpy((unsigned char*)&remoteaddr2.sin_addr,
                           (unsigned char *)rip, IP_ADDLEN);
                 remoteaddr2.sin_port = ECHO_PORT;

                 rv = connect(sock2, (struct sockaddr *)&remoteaddr2,
                                        sizeof(remoteaddr2));
                 if(rv == -1){
                        error = sockapi_get_last_error();
                        fprintf(fd, "\n SOCK2 connect : Error = %d\n", error);
                        goto endclient;
                }
                                                /* Send and Receive */
                for(j = 16; j < 32; j++){

                  for(i=0; i<100; msg[i] = j, i++);
                  rv = send(sock2, msg, 100, 0);
                  if(rv == -1){
                        error = sockapi_get_last_error();
                        fprintf(fd, "\n SOCK2 send : Error = %d\n", error);
                        goto endclient;
                  }

                  rv = recv(sock2, rcv_buff, 200, 0);
                  if(rv == -1){
```

```
                error = sockapi_get_last_error();
                fprintf(fd, "\n SOCK1 recv : Error = %d\n", error);
                goto endclient;
            }
            else{
                fprintf(fd, "\n ECHO : Received message = ");
                for(i=0; i<10; i++)
                    fprintf(fd, "%x ", rcv_buff[i]);
                fprintf(fd, "\n\n");
            }
        }

                                    /* close the connection */
        if(sock2>0){
                rv = closesocket(sock2);
                if(rv == -1){
                    error = sockapi_get_last_error();
                    fprintf(fd, "\n SOCK2 close socket : Error = %d\n",error);
                }
                sock2 = 0;
        }


                /***** ECHO CLIENT 3 :
    - blocking,
    - connected datagram without bind ****/

        sock3 = socket(PF_INET, SOCK_DGRAM, 0);
        if(sock3 == -1){
                error = sockapi_get_last_error();
                fprintf(fd, "\n SOCK3 socket : Error = %d\n", error);
                goto endclient;
        }
                                        /* no bind */

                                        /* connect the socket */

         remoteaddr3.sin_family = AF_INET;
         sockapi_dot2ip(rip, IP_ST2_ETH);
         memcpy((unsigned char*)&remoteaddr3.sin_addr,
                (unsigned char *)rip, IP_ADDLEN);
         remoteaddr3.sin_port = ECHO_PORT;

         rv = connect(sock3, (struct sockaddr *)&remoteaddr3,
                                    sizeof(remoteaddr3));
         if(rv == -1){
                error = sockapi_get_last_error();
                fprintf(fd, "\n SOCK3 connect : Error = %d\n", error);
                goto endclient;
        }
                                        /* Send and Receive */
        for(j = 32; j < 48; j++){
          for(i=0; i<100; msg[i] = j, i++);
          rv = send(sock3, msg, 100, 0);
          if(rv == -1){
                error = sockapi_get_last_error();
                fprintf(fd, "\n SOCK3 send : Error = %d\n", error);
                goto endclient;
          }

          rv = recv(sock3, rcv_buff, 200, 0);
```

```
                if(rv == -1){
                        error = sockapi_get_last_error();
                        fprintf(fd, "\n SOCK3 recv : Error = %d\n", error);
                        goto endclient;
                }
                else{
                        fprintf(fd, "\n ECHO : Received message = ");
                        for(i=0; i<10; i++)
                            fprintf(fd, "%x ", rcv_buff[i]);
                        fprintf(fd, "\n\n");
                }
                }
                                                /* close the connection */
                if(sock3>0){
                        rv = closesocket(sock3);
                        if(rv == -1){
                           error = sockapi_get_last_error();
                           fprintf(fd, "\n SOCK3 close socket : Error = %d\n",error);
                        }
                        sock3 = 0;
                }

        endclient :
                if(sock1>0){
                        rv = closesocket(sock1);
                        if(rv == -1){
                           error = sockapi_get_last_error();
                           fprintf(fd, "\n SOCK1 close socket : Error = %d\n",error);
                        }
                        sock1 = 0;
                }

                if(sock2>0){
                        rv = closesocket(sock2);
                        if(rv == -1){
                           error = sockapi_get_last_error();
                           fprintf(fd, "\n SOCK2 close socket : Error = %d\n",error);
                        }
                        sock2 = 0;
                }

                if(sock3>0){
                        rv = closesocket(sock3);
                        if(rv == -1){
                           error = sockapi_get_last_error();
                           fprintf(fd, "\n SOCK3 close socket : Error = %d\n",error);
                        }
                        sock3 = 0;
                }
            }
        }
```

Appendix D

Example of a datagram socket server

This example shows you how to implement a datagram server that echos the received bytes back to the client.

```
#define DGRAM_SRV_PORT   567

static void datagram_server()
{
    int                         srvsock = 0;
    struct sockaddr_insrvaddr;
    struct sockaddr_inclientaddr;
    int                         clientaddrlen = sizeof( struct sockaddr_in );
    int                         error;
    char                        buffer[1024];
    int                         received_bytes;
    int                         rv;

/***** DATAGRAM SERVER :
                                - blocking,
- non connected datagram ****/

        srvsock = socket(PF_INET, SOCK_DGRAM, 0);
        if(sock1 == -1){
                error = sockapi_get_last_error();
                fprintf(fd, "\n SRVSOCK socket : Error = %d\n", error);
                goto endserver;
        }
                                        /* bind to any local endpoint */

        srvaddr.sin_family = AF_INET;
        srvaddr.sin_addr.s_addr = INADDR_ANY;
        srvaddr.sin_port = DGRAM_SRV_PORT;

        rv = bind(srvsock, (struct sockaddr*)&srvaddr, sizeof(srvaddr) );
         if(rv == -1){
                error = sockapi_get_last_error();
                fprintf(fd, "\n SRVSOCK bind : Error = %d\n", error);
                goto endserver;
        }

       for( ; ;){

                / * receive datagrams on the socket from a Client */

           received_bytes = recvfrom(srvsock, buffer, sizeof(buffer), 0,
                                        (struct sockaddr *)&clientaddr, &clientaddrlen);
```

```
                if(received_bytes == -1){
                        error = sockapi_get_last_error();
                        fprintf(fd, "\n SRVSOCK recvfrom : Error = %d\n", error);
                        goto endserver;
                }

                    /* echo back to the Client the received bytes */

                rv = sendto(srvsock, buffer, received_bytes, 0,
                        (struct sockaddr *)&clientaddr, clientaddrlen);

                if(rv == -1){
                        error = sockapi_get_last_error();
                        fprintf(fd, "\n SRVSOCK sendto : Error = %d\n", error);
                        goto endserver;
                }
            }

        endserver :
          if(srvsock>0){
                rv = closesocket(srvsock);
                if(rv == -1){
                   error = sockapi_get_last_error();
                   fprintf(fd, "\n SRVSOCK close socket : Error = %d\n",error);
                }
                srvsock = 0;
          }

    }
```